An In-Depth Look into Cryptographic Hashing Algorithms

Jonathan Michael Krotkiewicz

Advisor: Dr. Mike Biocchi

April 1st, 2016

Abstract

Cryptographic hash functions, namely Message Digest 5 and Secure Hash Algorithm 1 were published over two decades ago and are still in frequent practice as a password security measure. Since publication, associated weaknesses and vulnerabilities have been identified with each function. From an information security perspective, the algorithms on their own are considered broken and insecure respectively. The presented literature seeks to illustrate the degree of vulnerability associated with the credited algorithms through extensive research, relevant statistical data, and firsthand experimentation. Well known attack methods such as a dictionary and rainbow table attacks are undertaken against a set of MD5 and SHA-1 hash values in a real environment to extract significant data regarding time and space complexity. The data are used in comparison to approximated results of secure cryptographic hashing standards in practice today. Consequently, information to counteract such attack methods is discussed in detail to proactively prevent the likelihood of a successful data breach in a real system.

Table of Contents

List of Figures

List of Tables

This page intentionally left blank

Chapter I Introduction

Today, a plethora of websites, systems, applications, and programs individuals operate on a daily basis employ a password authentication scheme to secure and protect data. The most commonly known method to secure passwords is through the use of cryptographic hashing algorithms. Primarily, these algorithms take plain text, which usually represents a password, and reconstructs the text into a string of characters which effectively mask's the original plain text. The issue with this approach is even if a system uses a cryptographic hashing algorithm, there is no guarantee the passwords will be protected. Many factors cause a variance between highly secure passwords and ones at risk, such as the selected cryptographic hashing and the strength of the overall password. Knowledge of this subject matter needs to be vastly considered and extensively researched primarily for developers implementing a password scheme into a proposed system.

1.1 Problem Definition

Cryptographic hash functions Message Digest 5 (MD5), and Secure Hash Algorithm 1 (SHA-1) were published almost over two decades ago as a means to secure passwords in the environment they are implemented. Since the publication of these mentioned algorithms, associated weaknesses and vulnerabilities have been identified. From a password security perspective, the algorithms on their own are considered broken and insecure respectively. Assuming the implementation of either MD5 or SHA-1 in a database storing the passwords of registered users, determining the plain text with the technological resources available today would be feasible. On the contrary, theses algorithms are still in frequent practice as a password security measure (See Appendix A). In today's current systems and services, which authenticate users via a username and password scheme, MD5 and SHA-1 in addition to deprecated and homemade cryptographic hashing functions should not be employed due to their security flaws.

1.2 Objective

This thesis aims to prove why MD5 and SHA-1 should not be used as a means to secure passwords based on in-depth analysis and experimentation. The following objectives will be fulfilled through extensive research to prove this statement. Prove MD5 and SHA-1 are not acceptable cryptographic hashing algorithms for current systems, investigate user password complexity to illustrate how to create secure, yet simple to remember passwords, and determine how to effectively store passwords on the developer’s end as a secondary defense mechanism. To further complement these objectives data will be gathered from firsthand experimentation regarding MD5 hash generation.

1.3 Thesis Overview

The subsequent chapters will cover the topics mentioned as follows. Chapter 2 discusses the negligence of consideration to password security, the brief history of cryptography, hashing in its most basic form, the differences between hashing and encryption, and digital signatures. Chapter 3 covers the discussion of passwords in regards to user complexity, an explanation to cryptographic hashing, MD5, and SHA-1, methods to compromise a hash value, a discussion on how to securely protect passwords, and results of compromised systems with MD5 or SHA-1 implemented. Chapter 4 discusses and presents information on the future of hashing algorithms and alternative means to secure passwords. Chapter 5 covers the project/experimentation with relevant results. Chapter 6 presents the conclusion of this thesis as well as further work on the subject matter. For this literature, the following sections in Chapter 2 will cover significant background prerequisite knowledge to understand the next chapters to a basic degree.

Chapter II History and Introduction of Cryptographic Hashing

When the Internet first came to be, the task of identifying users was not a prominent concern, but as more networks began to spread globally, user identification became an ever-ongoing problematic issue. Authentication and user identification today is a rising matter regarding challenges and complexity, especially with the vast number of individuals using the Internet daily. Without being able to identify someone for themselves, the trustworthiness of users online would be non-existent. Throughout the history of authentication in the past and today regarding the Internet, the most common form, which is still in extensive practice, is the use of passwords [7]. Without question, in the event of a password, the potential exposure of an individual's private information becomes present.

2.1 Security Negligence

During July of 2015, one of the largest information leaks of the year occurred (accredited by a hacker group known as "The Impact Team") where the infamous website known as *Ashley Madison*, was hacked resulting in over 20 gigabytes of data including user records released. This data also led to more than 11 million passwords revealed [7]. As of July, the website had roughly 39 million members, whereas currently the website has registered another four million after the breach.

This example exemplifies the negligence of individuals taking online security details into account, both on the developer and user end. If the website's developers were proactive and took the necessary precautions to their users' security, the integrity of their sensitive data would have remained secure. Although surprisingly, with the registration of millions of additional members after the hack this illustrates users were not aware of the security either breach or do not care about their online security even with the most private of matters. It is worth noting that the full credit numbers were not a part of the leaked data, which could have made the situation much more severe in the minds of the members. As the course of action employed by the Ashley Madison website administrators to improve their security is currently unknown; potential members should highly consider and question the website's current security details. This example is only one of many massive information leaks, which occurred throughout the last decade.

2.2 What is Cryptography?

According to the *Concise Oxford English Dictionary,* the definition of cryptography is “the art of writing or solving codes.” While this is accurate historically, the field of cryptography today is vast, both scientifically and mathematically. This definition was oriented around the codes used to convey secret communication used between two parties in the past, but today encompasses mechanisms for ensuring integrity, techniques for exchanging secret keys, protocols for authenticating users, electronic auctions and elections, digital cash, and more. Modern cryptography can be defined as “*the study of mathematical techniques for securing digital information, systems, and distributed computations against adversarial attacks”* [6].

Late in the twentith century, cryptography, as the original definition mentioned, was an art with little theory to make use. Thus, making or breaking new and existing codes relied heavily on creativity since there was no constitution on what was truly a “good” code. During the 1970’s and 1980’s cryptography took a drastic turn when theories were created allowing cryptography to be a subject of significant study involving both science and mathematics. This impact influenced how the computer security field was viewed by researchers [6].

The most significant differences between traditional cryptography in the past and modern cryptography today is its use. In the past, cryptography was primarily used by governments and military organizations, today, cryptography is applied in everyday life by most people everywhere without even know it. An example would be authenticating an individual on a website, a workplace system, using a credit card, and even a personal computer at home. The degree of cryptography use today is at the point where developers with relatively little experience are expected to incorporate cryptographic mechanisms in their applications [6].

As cryptography today is very mathematically and scientifically oriented, the intricate details on the workings of the mentioned algorithms will not be the focus of this thesis, but are lightly discussed.

2.2 What is Hashing?

Hashing plays a very significant role in information security regarding passwords. The foundation for cryptographic algorithms are built upon this concept. Basic preliminary knowledge of hashing will be discussed to act as supplementary material for comprehension.

In computer science, hashing is the process of taking a sequence of characters (string), known as the key, and then transforming that key to a usually shorter fixed-length sequence of characters, known as the hash value [5]. To accomplish this process, an algorithm called a hash function uses the input as the key, and the output is the hash value or hash for short. Simply, the benefit and main use of hashing is time reduction. Instead of searching for the entire original string in a database, for example, just the hash value itself needs to be searched thus minimizing the amount of resources required for a search query [5]. Figure 2-1 illustrates hashing in its simplest form. As mentioned, faster data retrieval is one of the primary uses for hashing, while also, it can be used for encryption and cryptography of passwords.

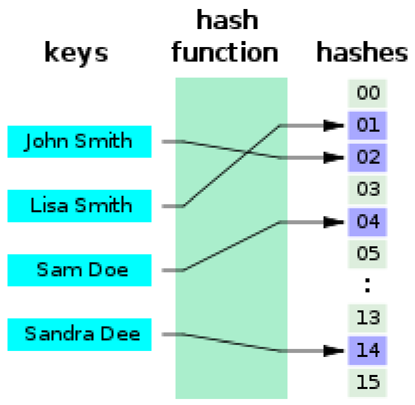


Figure 2-1 A High Level Illustration of Hashing [48]

The most important idea to grasp about hashing is the hash function should map each key to a unique hash value ideally [4]. In other words, every single string matches up to only one hash, meaning if the same input to the hash function is used multiple times, the resulting output will always be the same. Furthermore, a key should not have the resulting hash of another key. This event is known as a collision which is discussed in Chapter 3. If the mentioned criteria are met, the hash function is known as a perfect hash function [4]. With respect to cryptography, building a perfect hash function for passwords is a very complex and difficult undertaking, as many years are required to complete a single function. The closer one of these functions gets to being perfect, the security factor of the function increases as a result.

As many hash functions exist for different purposes with various inputs and outputs, the same intention as the criteria mentioned above is the basis for every algorithm. With this said, a particular hashing function may work well for database operations, although, more than likely will not work error checking or cryptographic purposes [5]. In conclusion, of the discussion on hashing, every cryptographic hashing function is a hashing function itself, but not every hashing function is a cryptographic one.

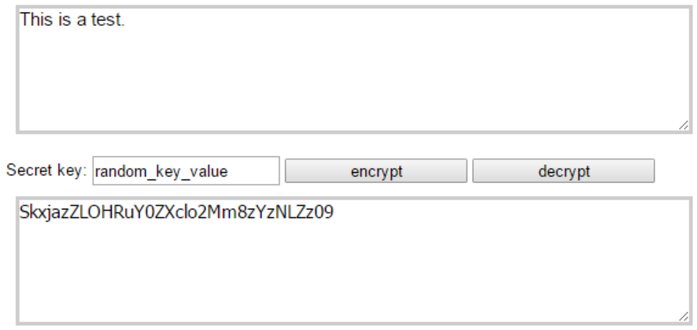
2.3 Encryption vs. Cryptographic Hashing

As encryption and cryptographic hashing (mistaken as just hashing) are often misunderstood and used interchangeably with regards to information security, the comprehension of both is important throughout the presented information on cryptographic hashing. The fundamental relation between the two are the intentions to secure data, but are differentiated by the inputs, outputs, use, and operation of their respective algorithms to carry out their tasks. Understanding the differences between the two is very preliminary when addressing either's subject matter.

Encryption is the process of transforming data or information from its plain text form into a product, which is unreadable and meaningless where the result can only be undone through the possession of special knowledge, known as the encryption key. In contrast, to hashing itself, encryption at its most basic level is performing the same task as hashing, being transforming data into another form, although with hashing the resulting data is typically in a structure much lengthier than the origin string and impossible to reverse.

The plain text is simply raw data ranging from text, documents, messages, to confidential data stored in databases. To carry out the process of encryption, a specialized algorithm known as a cipher is used. The encryption key is a string of characters chosen and used to encrypt the data and decrypt the data back to its origin. In short, the key provides initial information required by the cipher. The use of the encryption key can be seen in Figure 2-2.

There are a wide variety of ciphers available where their security details range from very insecure to highly secure. A modern cipher used today is known as advanced encryption standard (AES). Thus, the selection of a cipher has a direct relation to the security of data provided in return [1]. As illustrated in Figure 2-3 the original message is transformed into a secure state through encryption, then returned to its origin through decryption on the other end.



This is a test.

Secret key: random_key_value [encrypt] [decrypt]

SkxjazZLOHRuY0ZXclo2Mm8zYzNLZz09

Figure 2-2 An illustration of using a key for encryption

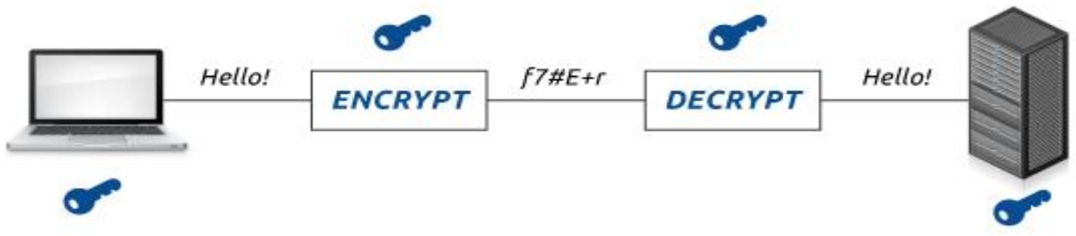


Figure 3-3 A high level illustration of encryption [49]

Cryptographic hashing shares the property of transforming data or information from its plain text form into a product, which is unreadable and meaningless, but unlike encryption, there is no key used to undo the result. The goal is simply accepting a message as input then produce an output (the hash value) no individual can understand. Without being able to easily understand what a hash value's plain text is, the user's password remains confidential to him or her only. With basic hashing, the hash value of the function is not designed with the intention of being irreversible, but with cryptography in mind, it is the utmost important aspect of the function itself.

The algorithms used for encryption are known as ciphers while the algorithms for cryptographic hashing are known as cryptographic hash functions. Regarding implementation of the two, both are very dissimilar with their specifications, and design. Another similarity to encryption is many cryptographic hashing algorithms are available today, and the choice of algorithm is again directly related to the security provided in return. Two examples as mentioned are MD5 and SHA-1.

With subsequent login attempts after account creation, a user will enter their password as prompted where the plain text string is used as input to the cryptographic hash function on the server's end.

Once the hash value is generated it is compared to the hash value stored in the server's database. Through this, the user's password is still private, but provides a means to authenticate them.

The most effective means to gain a clear and concise understanding of the differences between the terms is through the situations they are required. In a scenario where two people want to exchange messages over a distance securely, they would use encryption as it allows a message to be encrypted by one person and decrypted by the other with the utilization of a secret key. Through this, both the sender and receiver have a means to share messages securely while consequently the message cannot be intercepted by an external source then decrypted. Thus, encryption is a two-way street metaphorically speaking.

Cryptographic hashing, on the other hand, is a one-way street, where once a message is converted into an unreadable form, it cannot be reversed back to its original form regardless of the agent providing the plain text to be hashed. The most suitable example for cryptographic hashing is securing passwords. When implementing a database used to store information about users, a password is usually one part of the information needed to be stored. Instead of storing the plain text of the user's password, the hash value produced by the cryptographic hashing algorithm is stored instead. The user's password, as a result, is masked. Anyone using the database will only see a meaningless set of strings in the password field, which cannot be reversed to its origin. Thus, a user is the only person who knows their password. As seen in Figure 2-4, one the hash value is generated there is no method to undo the result.

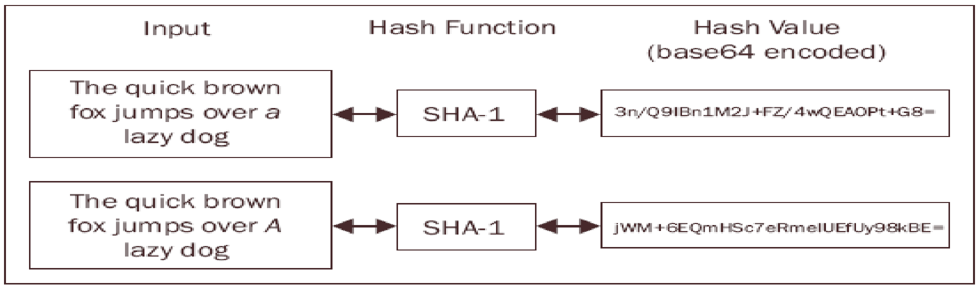

Input
Hash Function
Hash Value (base64 encoded)
The quick brown fox jumps over a lazy dog
SHA-1
3n/Q9IBn1M2J+FZ/4wQEAOPt+G8=
The quick brown fox jumps over A lazy dog
SHA-1
jWM+6EQmHSc7eRmelUEfUy98kBE=

Figure 2-4 An example of the hashing process

2.4 Checksums

In environments such as air traffic control and hospitals where safety is a very critical concern in the line of work, an incorrect entry of a number could potentially result in a harmful outcome in a particular task or patient, respectively. To mitigate or completely remove the possibility of error, the use of checksums can be employed. Checksums can be thought of as redundant strings related to the set of numbers to-be-entered (in this scenario) in such a way that the correctness of the checksum can be validated as opposed to checking all the numbers individually [50]. The criteria for such a method would require a unique output for every unique input for validity to be credible. Based on the current understanding of cryptographic hashing algorithms, one could be used to meet this particular need. Thus, they apply to be employed in this manner. For example, if hundreds of numbers were to be entered by a certain individual, they could hash the original set using one of the algorithms then after entry, they would hash the entered set. Through comparing the two hashes to one another, an error could be easily identified if the hash values were different.

In addition, checksums are commonly used when transmitting data over a network or the Internet. As data can be corrupted or intercepted through transit, the sender may consider generating the hash value of the data to be sent. Once the intended receiver obtains the data, they could hash it, then compare the value to the original hash value generated by the sender. The receiver could then determine if any problems occurred during transmission if their hash value did not match the sender’s value.

2.5 History and Introduction of Cryptographic Hashing Conclusions

In short, this chapter discussed an example illustrating how individuals and developers are negligible with information security regarding passwords, as even after the event of millions of passwords being compromised, millions of users in the following few months still registered for the breached website. Furthermore, the history and a standard definition of cryptography, a high level explanation of hashing and its applicability, the differences between encryption and cryptographic hashing, and how cryptographic hashing can apply to checksums was presented as well. This brief introduction should be adequate for a novice to grasp the concepts in the next chapter, as well as to gain a clear and concise comprehension of the intricacies relating to password storage, and the method used to obtain passwords from their hash values.

Chapter III Current Security

This chapter will be covering cryptographic hashing functions with their associated vulnerabilities primarily focusing on MD5 and SHA-1. Alternatives for replacement in addition to approaches to compromise password hashes such as rainbow tables and lookup tables are discussed in detail. Password types, salting and peppering passwords, and acceptable means to protect user's passwords are explored as well.

The advanced knowledge will be a clear and concise understanding of how passwords are hashed, methods on how password hashes can be obtained, and the difference between strong and weak passwords. Furthermore, the approaches hackers take to crack passwords, reasoning behind why MD5 and SHA-1 are no longer suitable cryptographic hashing algorithms, and methods of developing secure passwords which would effectively protect a user will be discussed. This information could greatly benefit both a novice developer and individuals without a thorough understanding of hashing who are required or expected to implement security protocols for passwords into an application or system.

3.1 Passwords

Before the discussion of cryptographic hashing algorithms, information on the topic of passwords needs to be explored to obtain a deep comprehension regarding the security effectiveness of a given cryptographic hashing algorithm with a candidate password.

3.1.1 Password Requirements

As known to most, a password is a secret word or phrase consisting of a string of characters used to gain access to various personal accounts and devices such as website user accounts, computers, network locations, documents, folders, hard drives, and mobile devices with the intention of securing data. Typically, a password should be complex enough to keep it a secret while being simple to remember. On the other hand, the use of static passwords today can be seen as a security flaw due to the ever-increasing need for the Internet. The reasoning is if a user applies the same password throughout various accounts where a hacker obtained their password, that person's data, including emails, documents, social networking accounts, and much more could be severely compromised [2]. Without question, protecting all of this data is a great security need where passwords remain the most well known form of authentication and verification [11]. Essentially a person's password is equivalent to their house key, it is the most prominent factor for online security of data when using the Internet.

When the Internet was first becoming accessible to the public, requirements for passwords were much different than they are today. Previously, systems only required (or allowed) a six to eight characters of mixed alpha passwords which was considered adequate. In contrast, today some applications have their requirements for passwords being twelve to fourteen characters long with the combination of lowercase and uppercase letters, numbers, and special symbols to be considered effective [11].

After analyzing the requirements of the most well known websites/services such as Facebook, Yahoo, Gmail, Hotmail, and Twitter, there is a consensus for the requirement standards of passwords.

- Passwords must be a minimum of eight characters.
- Passwords can be a maximum of 16 characters and upwards.
- Passwords are recommended to have at least one upper case and lower case letter, number, and special symbol, but are not required by some of the websites.
- Passwords previously used are not recommended or allowed.

From a security perspective, the requirements are satisfactory, but not sufficient. The reasoning is in the event a database is compromised, the weakest password hashes stored will be revealed first, thus leaving the individuals who chose the bare minimum of the requirements most vulnerable. A password might be considered adequate by the system, but not considered adequate in the situation of a potential attack.

3.1.2 Weak Passwords

"Selecting a weak password is like closing your front door but not locking it. A password is weak if it can be guessed easily" – United States Computer Emergency Readiness Team [12]

Social media, messaging, online banking, e-mail, along with various other online applications utilized by users, entering sensitive data can potentially be misused if acquired from a third party source. With this possibility, one would expect users to create very secure passwords; however, it is commonly known users comprise passwords based on simple and easy to remember combinations of words consisting of only alphanumeric possibilities [10]. The *International Journal of Human-Computer Studies* describes this as “The password problem” [3]. Specifically, the problem states two requirements which conflict with one another and are unable to be satisfied by users due to the capacity of human memory.

1) Passwords should be easy to remember, and the user authentication protocol should be executable quickly and easily by people.
2) Passwords should be secure, i.e., they should look random and should be hard to guess; they should be changed frequently, and should be different on different accounts of the same user; they should not be written down or stored in plain text.

When a person is asked, what they believe is a weak password, the first thought is usually passwords such as “abc123,” “password,” “qwerty,” or any other of the most commonly used passwords (Appendix B). These passwords are indeed weak, but a password of short length, typically less than ten characters, and with the absence of mixed alphanumeric-symbol character combination is considered less than desirable. The reasoning behind this is simply because these passwords will be the most vulnerable in the event of an attack making them a targeted priority. A potential attacker will concern themselves with the weakest passwords first as they require the least duration of time to crack. Passwords revealed the earliest can be seen as weak with respect to the length of time needed to reveal them. For instance, if a password is twenty characters long and revealed first with the tradeoff of taking over a year to crack, that password can be seen as incredibly secure in comparison to an eight character all lower case password which would take less than a day to obtain.

Based off the number of possibilities for a given character set, one can easily see how the complexity factor of strength for a particular password increases by adding a mix of different types of characters. This relation is shown in Table 3-1 where ‘n’ represents the length of a password. Thus, there is a correlation between weak passwords and time where a password’s strength can be measured based on the duration of time required to obtain it. Unfortunately, this correlation is a resulting ramification of cheaper and more powerful technology/hardware being created each year with respect to Moore’s law. Consequently, the derived effect is password strength decay over time. This correlation will be implicitly understood in section 3.3. Eventually, a password, which was once considered strong, will slowly decay to the point of ineffectiveness after a certain duration of time directly based on the complexity of the given password. Simply, a password of fifteen characters, for example, will decay exponentially faster in strength then one of which is twenty characters. With this said the cryptographic hashing algorithm utilized plays a very significant role in the strength of a given password. Later in this chapter, it will be understood a password is only as effective as the cryptographic hashing algorithm used to mask it.

Character Set (The associations can be seen in Appendix E)	**Number of Possibilities**
Numeric	10^n
Only Alpha or Lower Alpha	26^n
Only Symbols-Space	33^n
Only Lower Alphanumeric or Alphanumeric	36^n
Mix alpha	52^n
Mix alphanumeric	62^n
Mixalphanumeric-Symbol32-Space	95^n

Table 3-1 The number of different combinations for the various character sets

3.1.3 Strong Passwords

The password requirements listed earlier specified, it is sometimes required to include at least one upper case and lower case letter, a number, and a special symbol to add to the overall password strength. While this is true based on the number of possibilities for the given character set, it is not the only alternative to increasing the strength of a password. Simply, the utmost strongest of passwords are based on their length, not the variation of different characters. Although, adding a variation of characters for a password of a great length will increase the strength further exponentially. For example, a user password representing a memorized sentence of 20 characters of all lower case letters would yield a result of 26^{20} combinations to obtain. This example, provides a far greater number of possibilities in comparison to an eight-character password containing a mix of all the character types being 95^8 possibilities. In other words, the longer the password, the greater the strength, even if it only consists of lowercase letters.

Furthermore, the difference of strength between the two mentioned passwords, in this case, can be easily understood by thinking about the duration of time a computer would require to reach the given strings based on their length chronologically. The twenty-character password will take an exponentially longer duration to reach than that of the eight-character password. In addition, potential attackers will only concern themselves with the weakest password hashes stored, thus, a very long password of all lower case letters will not be targeted before the passwords of a smaller length.

With respect to the password requirements, a password consisting of the mix of different case letters, numbers, and symbols, with a reasonable length is still very secure in comparison to weak passwords.

3.1.4 Creating Memorable but Strong Passwords

Longer passwords are indeed stronger in terms of time complexity, but a password being the letters ‘a’ through ‘z’ would not be secure due to its predictability. In order to make memorable, but secure passwords, passphrases can be utilized. Based on an example provided by Microsoft, start with a sentence or two: “Complex passwords are safer.” Remove the spaces between the words in the sentence: “Complexpasswordsaresafer.” Turn words into shorthand or intentionally misspell a word: “ComplekspasswordsRsafer.” Add to the length with numbers, which are meaningful after the sentence: “ComplekspasswordsRsafer2011” [12].

An alternative approach to complex and strong password creation is through a method created by psychologists called a “PsychoPass.” As this alternative is more reliant on mental practice, the tradeoff is a secure and complex password. “Rather than thinking of a password, a person only needs to think of an action sequence” [13]. According to Pietro [13], the method consists of four steps, which are shown below, and the illustrations can be seen in Figures 3-1 and 3-2.

1) Begin with a letter on the keyboard.
2) Memorize a sequence of actions. (Something like “the key on the left, then the upper one, then the one on the right”, and so on)
3) Memorize the sequence. (not the letters used)
4) Create as many passwords as you want by remembering only the first letter and the sequence. Using different types of sequences, it is possible to generate thousands of different passwords. Using sequences' combination is possible to create an infinite number of passwords. Moreover, the created passwords will be a nonsense sequence of letters, numbers, and symbols, resilient to any attack.

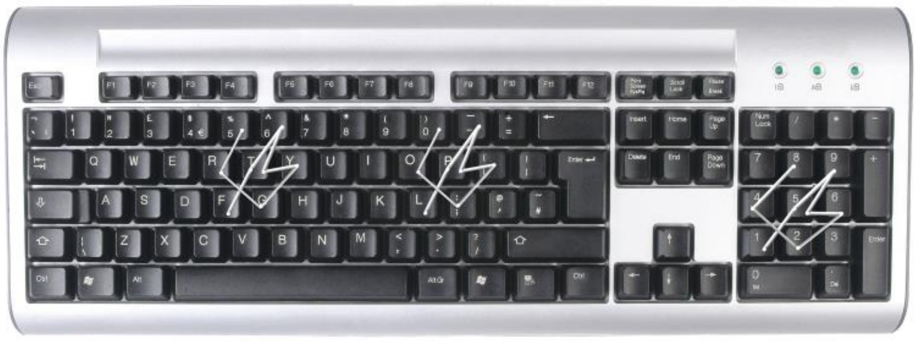


Figure 3-1 PsychoPass example 1 [13]

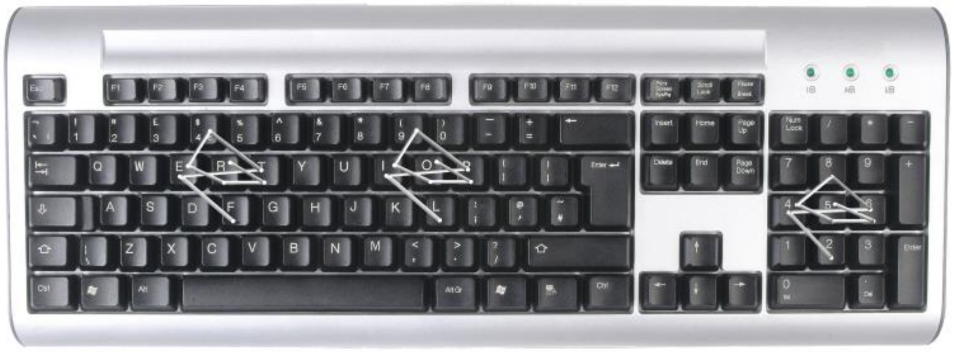


Figure 3-2 PsychoPass example 2 [13]

3.1.5 User Password Complexity Studies

With the comprehension of the differentiation between weak and strong passwords, the discussion on this topic can close with the analyzation and results of surveys and statistical analysis on user-generated passwords in practice. Some of these results are based on cracked passwords from previously hacked websites. All results are within the last seven years.

Survey 1: Password in Practice: A Usability Survey

In 2011, a usability survey was published by the *Journal of Global Research in Computer Science* (JGRCS) to study and investigate passwords specifically referring to memorability, password-based security, and the problem of forgetting passwords, alternative authentication schemes, and the usability and security of alphanumeric passwords. The investigation of user password practices and the methods users apply to construct passwords are the focal point of the developed survey. The Figures 3-3 to 3-9 are taken directly from the survey to illustrate the password habits of the 195 participants from ages 17 to 61 years, where 76% were male, and the other 24% were female [10].

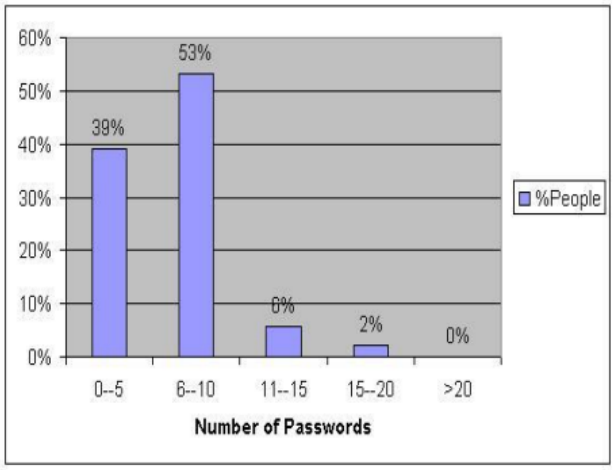


Figure 3-3 Number of participant's passwords for various accounts [10]

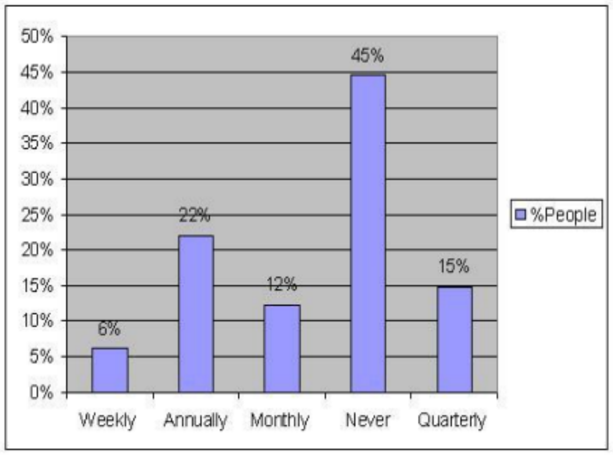


Figure 3-4 Frequency of users changing their passwords [10]

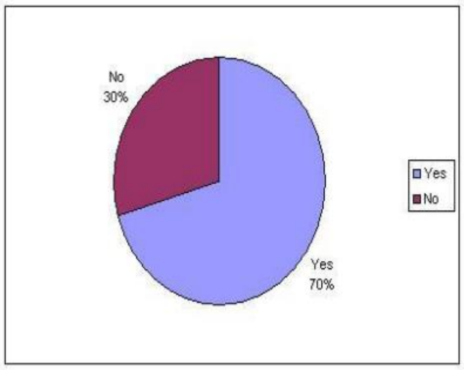


Figure 3-5 Response on switching back to an old password [10]

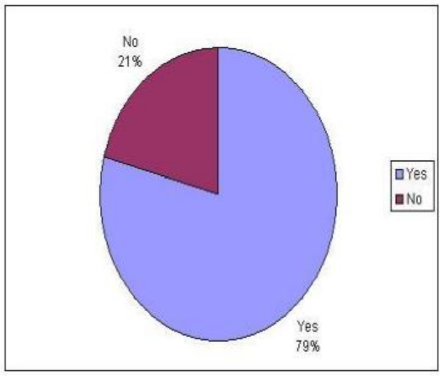


Figure 3-6 Response on users keeping the same password for multiple accounts [10]

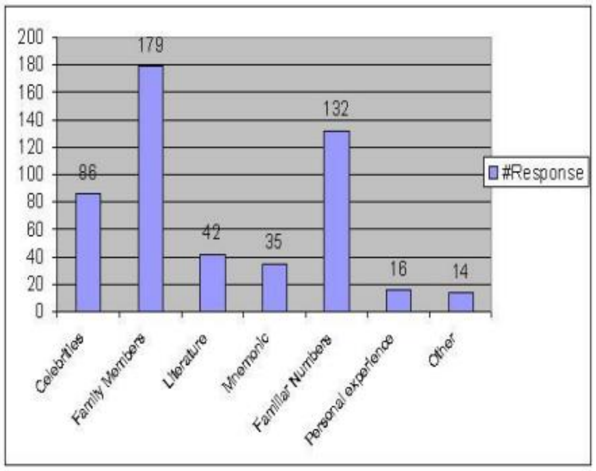


Figure 3-7 Sources of password inspiration that participants use [10]

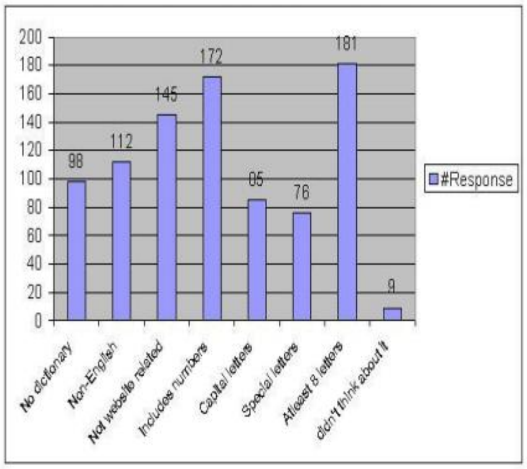


Figure 3-8 Factors considered by participants during password creation [10]

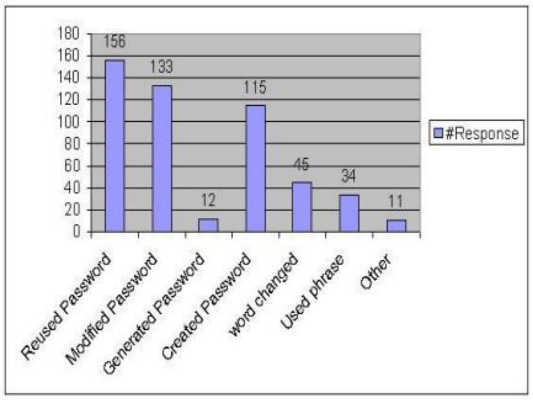


Figure 3-9 Response of participants on new password creation (note: created password is based on a name) [10]

After analyzing these results firsthand, the common trend for a majority of user's password habits based on this survey are as follows [10]:

- Users never change their passwords.
- When users change their passwords, they revert to a previously used one.
- Users keep the same password for multiple accounts.
- User passwords are based on family members, celebrities, and familiar numbers.
- User's passwords are based on the factors of at least eight letters, include numbers, and not related to the website.
- Users create new passwords through reusing old passwords, a modification of an existing password, and the creation of a new password based on a name (user's name, pet's name, or significant other's name) or date.

Furthermore, according to the conclusions of the survey, users who are experienced with multiple passwords write them down while inexperienced users keep the same password for multiple accounts and remember them based on memory. Users also seem to adapt password management schemes such as using the same password for multiple websites [10].

Survey 2: Text Entry Method Affects Password Security

The second survey undertaken in 2014 was a study performed at Rutgers University, where groups of participants were tested to determine the complexity and types of passwords generated by the three primary methods for text entry being a laptop, tablet, and smartphone. After the group had created three different passwords for each of the devices, attacks were performed on the passwords using "John the Ripper," and "Hashcat." Out of the 189 passwords among 63 participants generating a password, for each entry method 24 (38.1%), 24 (29.6%) and 16 (35.6%), respectively were cracked [15]. The results from the complexity of the passwords are illustrated in Figure 3-10 and Figure 3-11.

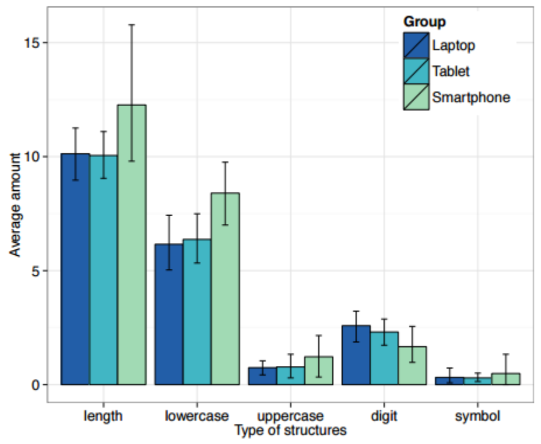


Figure 3-10 Participant's password statistics chart 1 [15]

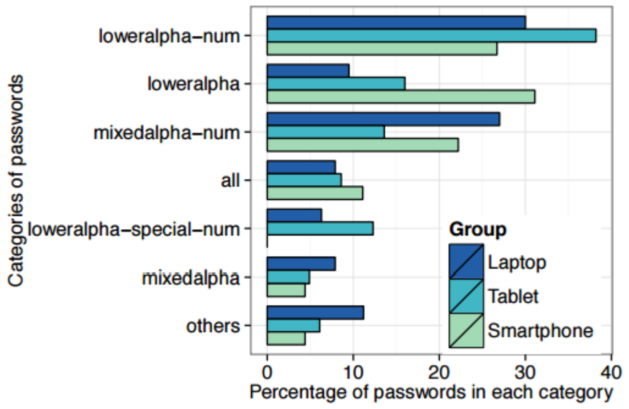


Figure 3-11 Participant's password statistics chart 2 [15]

As shown, the majority of the passwords were standard, they lacked a mixture of digits, symbols, upper/lowercase letters, and were of a less than optimal length. These results clearly portray the majority of user passwords in this study are considerably weak based on the discussion earlier.

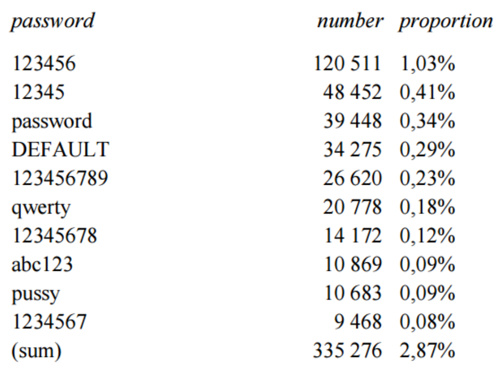
Statistical Analysis 1: Ashley Madison Hack

The statistical analysis on user-generated passwords created based on the hacked database in 2015 from the website Ashley Madison[1] is shown as follows. According to the results from the Ashley Madison users, "123456" was the most frequently used password, over 120 thousand users (1.03%) out of the 11.7 million whose passwords were successfully cracked used it [7]. The top ten passwords from the database are shown in Table 3-2, and the character sets used by the users are presented in Figure 3-12.

According to the list of the top 25 passwords of 2015 (Appendix B), most of these strings from Table 3-2 were present in this list. To complement the second survey from Rutgers University in

[1] Ashley Madison: www.ashleymadison.com

2014, Figure 3-13 shows a direct relation between the results where the largest majority of accounts for the website consisted of primarily all lower alpha and lower alphanumeric passwords.



password	number	proportion
123456	120 511	1,03%
12345	48 452	0,41%
password	39 448	0,34%
DEFAULT	34 275	0,29%
123456789	26 620	0,23%
qwerty	20 778	0,18%
12345678	14 172	0,12%
abc123	10 869	0,09%
pussy	10 683	0,09%
1234567	9 468	0,08%
(sum)	335 276	2,87%

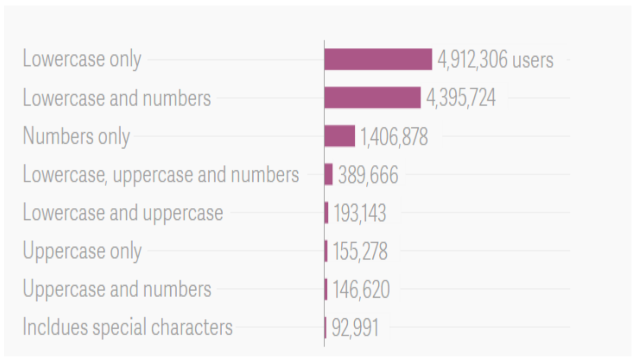
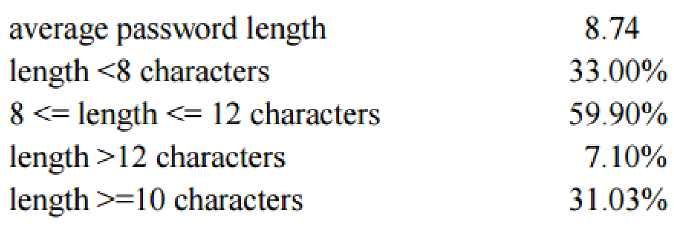


Table 3-2 Top ten Ashley Madison passwords [7] *Figure 3-12 Ashley Madison user password character set use [16]*

Statistical Analysis 2: Rockyou.com Hack

In 2009, the website RockYou[2] was hacked through Structured Query Language (SQL) Injection resulting in the public display of 32 million passwords, which were stored in plain text. Inevitably, with this implementation every user was at risk as anyone could see his or her password. This example exemplifies the catastrophic aftermath of storing the plain text of user passwords. In Keszthelyi's paper "About Passwords" [24], he downloaded the list and of the 32 million, slightly over 14 million passwords were unique and close to two thousand were over thirty-two characters in length. Table 3-3 illustrates the original statistical data Keszthelyi has generated from the password list in regards to the password length of the users.



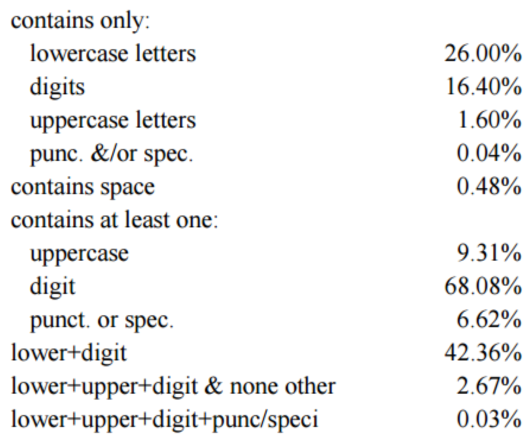
average password length	8.74
length <8 characters	33.00%
8 <= length <= 12 characters	59.90%
length >12 characters	7.10%
length >=10 characters	31.03%

Table 3-3 Rockyou.com password statistics [24]

The second table (Table 3-4) from the paper shows the statistics based on the different characters used in the passwords where it turns out over a quarter consisted of only lower case letters. "Digits

[2] RockYou: www.rockyou.com

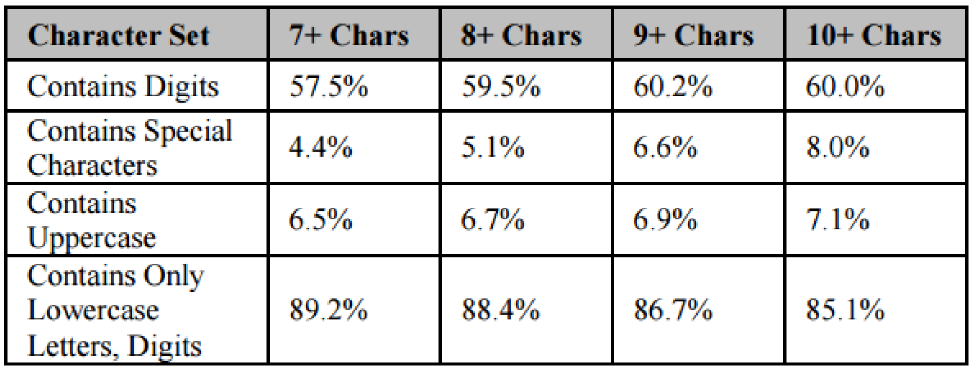
are preferred to uppercase letters or others; two third of the passwords contain digits while only about 16% of them contain uppercase letter(s), or punctuation mark(s), or another special character(s). The results meant most people do not follow the general rule of passwords, which a password must contain all kind of character types" [24].



contains only:	
lowercase letters	26.00%
digits	16.40%
uppercase letters	1.60%
punc. &/or spec.	0.04%
contains space	0.48%
contains at least one:	
uppercase	9.31%
digit	68.08%
punct. or spec.	6.62%
lower+digit	42.36%
lower+upper+digit & none other	2.67%
lower+upper+digit+punc/speci	0.03%

Table 3-4 Rockyou.com password statistics [24]

Another author by the name of Matt Weir also did a statistical analysis of the cracked website in his paper "Testing Metrics for Password Creation Policies by Attacking Large Sets of Revealed Passwords" [41]. His paper revealed a significant portion of user's passwords contained only lower case letters and digits, regardless of the provided password lengths. With only seven characters consisting of a lower alphanumeric mix, these passwords (if hashed) would be at risk of being compromised before every subsequent password (Table 3-5). The complete statistical analysis can be seen at [41].



Character Set	7+ Chars	8+ Chars	9+ Chars	10+ Chars
Contains Digits	57.5%	59.5%	60.2%	60.0%
Contains Special Characters	4.4%	5.1%	6.6%	8.0%
Contains Uppercase	6.5%	6.7%	6.9%	7.1%
Contains Only Lowercase Letters, Digits	89.2%	88.4%	86.7%	85.1%

Table 1-5 Password information from the rockyou1 list [41]

3.1.6 Password Conclusions

Based on these results, a significant conclusion can be established for the general population of users with a password. A majority commonly creates weak passwords and use irresponsible password selection methods. With only lower alpha characters being present in a given password of a short length, the nature of the minimized number of possibilities results in the string to be more vulnerable in contrast to others. By having a combination of letters, symbols, digits, and a significant length of characters, or a very long password of all lower case letters, the risk of vulnerability can be greatly reduced. Throughout the next few sections, the severe risk of using weak passwords with specific cryptographic hashing algorithms will be understood. It is worth mentioning, as the cryptographic hashing algorithm implemented in a system is typically unknown to a user, due to security protocols, an individual should be wary of choosing their password for a website, application, or system.

3.2 Cryptographic Hashing

This section begins with a clear and concise explanation of cryptographic hashing at a relatively high level along with the specific requirements such algorithms must fulfill to be considered for application. Following will be an explanation of the algorithms MD5 and SHA-1 as the topic of interest, along with their associated known weaknesses.

3.2.1 Introduction to Cryptographic Hashing

Cryptographic hashing functions accomplish the goal of generating hash values, also known as the message digest. These functions map a string of arbitrary length to a hash value by transforming the input string to a unique output string [20]. For instance, if the input string was “administrator” the digest would look something like "200ceb26807d6bf99fd6f4f0d1ca54d4,” dependent on the function chosen. As mentioned previously, these algorithms are one-way functions; under no circumstance, should there be an inverse function. With an ideal cryptographic hashing function, reversing the digest back to the message should be infeasible and extremely difficult.

To explain the concept on a higher level imagine taking an apple than turning the apple into apple juice through a series of steps, then afterward attempt to turn the apple juice back to the apple as a whole. This task would be extremely infeasible and near impossible. Now imagine the apple as a password, the apple juice would be the resulting hash value of the function as it turns the password into a form that should be impossible to reverse. Figure 3-13 illustrates this example.

Figure 3-13 High level illustration of cryptographic hashing

One of the primary uses of these algorithms is for authentication purposes for a web service, application, or system, which is typically performed through a username/password pair where a user's credentials are compared to the records in a database to validate that user. The two approaches for storing a password are through either plain text or the hash value. Web services, which email their users a temporary password after account creation, store them as plain text where other services store the hash value. Storing plain text is a simplistic approach, although extremely insecure. Security should not be traded for simplicity in this regard. If a hacker were to obtain the contents of a database, they would be able to see all the user's passwords, as they are unmasked. The difference between storing plain text and hash values can be seen in Table 3-6 where the left side is a plain text storage representation while the right side is a representation of hash value storage. A user need not enter the hash value string as their password when the original password is entered for a website; the text goes through the algorithm transforming it into the hash value, which is then compared to the value stored in the password field.

Username	**Password (Plain text)**	**Username**	**Password (Hash Value)**
Joe	Ja7Dh2wh	Joe	2b5d95a2bac09abecb5248d85c39d275
Bob	Ld098sdhj	Bob	4bdd5c88198158829b30eae6493d3ea5
Alice	Jkfuifmasj	Alice	ed865cf28cb2598bb758e7fa3ab3f5da

Table 3-6 Storing plain text vs. storing hash values

Password hashing can be thought of as a second line of defense where the goal is to prevent an attacker from easily accessing user accounts. As seen in Table 3-6 the password hashes are meaningless and cannot be easily applied to determine the original plain text. Specifically, the hash values were generated using MD5 in this example. From a first glance, a person may think this is a very secure masking as they are unable to reverse the hash, although the effectiveness of how secure the hash values are is dependent on the cryptographic hashing algorithm used for implementation. This relation will be seen in the next section. Through implementing a robust second line of defense for protecting clients against attackers regardless of the application, a person’s password can be effectively secured and made infeasible to determine given the hash value. Effectively securing passwords on a backend system is discussed in section 3.4.

3.2.2 Security Requirements for Cryptographic Hash Functions

As a warning to the reader, the creation of a cryptographic hash function is anything but a simple task. Many years of planning, design, and testing from the leaders and professionals in the cryptography field are required to make such a function. Not only is the series of steps a function takes to create a single digest extremely convoluted and heavily based on mathematics, science, and compression techniques, to say at the least, the process of creating a function is far more complicated. With this said, an individual should not attempt to set up their own function with the intentions of implementation to secure passwords in a given system under any circumstance. Following are the security requirements any cryptographic hashing function should satisfy to be considered safe to use in real systems and environments.

Some common and useful terminology: if H is a hash function, m is an input bit string, and h is the output of H applied to the input m, then we write h = H (m). If h = H(m) then h is called the "hash" of m, m is referred to as a "preimage" of h, for a given input m, a "second preimage" of m is a different input m' such that H(m) = H(m'), if m and m' are different inputs such that H(m) = H(m') then the pair {m, m'} is called a "collision" for H [20].

A cryptographic hash function should satisfy the following criteria:

- (**Practicality**) computing the hash h (m) of any input m can be done efficiently,
- (**Preimage resistance**) given h, it is hard to compute a preimage of h, i.e. it is hard to compute an m such that h = H (m),
- (**Second preimage resistance**) given m, it is hard to compute a second preimage of m, i.e. it is hard to compute an m' such that m ≠ m' and yet H (m) = H (m'),
- (**Collision resistance**) it is hard to compute a collision for H, i.e. it is hard to compute m and m' such that m ≠ m' and yet H (m) = H (m') [20].

The requirement of practicality simply states the function should be able to compute hash values without overhead in a manner where there is minimum wasted resources or expense to achieve maximum productivity. Preimage resistance states: Given a hash value such as “200ceb26807d6bf99fd6f4f0d1ca54d4,” it is infeasible to determine the original input message being “administrator” in this case. The third requirement, second preimage resistance states: Given a message such as “administrator” it infeasible to determine another message such as “user” where both of the messages hash to the same output. The last requirement, collision resistance states: It is infeasible to determine two messages, which hash to the same value as their output. A collision at a high level can be seen in Figure 3-14. The difference between the third and fourth requirement is subtle, yet very significant. The primary difference is given a message to work with as opposed to starting with no resources respectively to determine the messages.

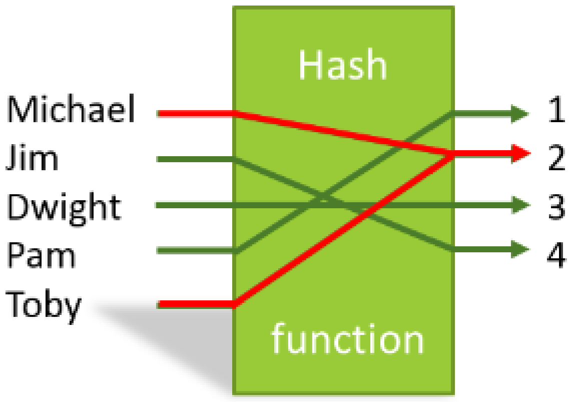


Figure 3-14 A High Level Hashing Collision

3.2.3 Message Digest 5

In 1992, a cryptographic hash function known as MD5 was designed and created by Ronald Rivest with the intention of improving MD4 as the algorithm was severely compromised [19]. MD5 along with MD4 belong to the series of message digest algorithms where the subsequent algorithms were designed and created to replace the predecessors. The output specifications of the algorithm are a 32 digit hexadecimal number being a 128-bit (16-byte) hash value. Primarily the algorithm is based on 32-bit integers with addition and bitwise operations such as XOR, OR, AND, bitwise rotation and Add (mod 232) [32]. A decade ago, MD5 was one of the most popular cryptographic hash functions in use; however today from an information security perspective, the algorithm is less than applicable for the application of cryptographic purposes due to a primary weakness. Example MD5 hashes are illustrated in Table 3-7. Appendix C is an overview of the MD5 algorithm.

Plain text	**MD5 Hash Value**
Password	5f4dcc3b5aa765d61d8327deb882cf99
PASSWORD	319f4d26e3c536b5dd871bb2c52e3178
Password	dc647eb65e6711e155375218212b3964

Table 3-7 The mapping between a plain text and its md5 hash value

Known Weaknesses of MD5

In 1993, a year after MD5's creation, weakness in the algorithm was identified by B. den Boer and A. Bosselaers, who found a "pseudo-collision" consisting of the same message with two different sets of initial values [32]. Despite this, no hard evidence was produced until 2004 where collisions were found [20]. A specifically designed attack, known as *modular differential* created by Xiaoyun Wang and Hongbo Yu, obtained collisions with MD5 in 15 minutes to an hour of computational time [19]. This attack is not only exclusive to MD5 but can be used to break other functions such as HAVAL-128, SHA-0, and RIPEMD. The attack, in general, is very convoluted as an extensive understanding of the intricate details of how MD5's algorithm performs is required. Regardless, the results proved to find two pairs of collisions for MD5, concluding its weakness to collisions, which turned out to be extremely feasible. Since then many collisions of MD5 have been found, one of which is shown in Figure 3-15 where the two message blocks hash to the same value being 79054025255fb1a26e4bc422aef54eb4. Full details on the attack can be seen at [19].

```
d131dd02c5e6eec4693d9a0698aff95c 2fcab58712467eab4004583eb8fb7f89
55ad340609f4b30283e488832571415a 085125e8f7cdc99fd91dbdf280373c5b
d8823e3156348f5bae6dacd436c919c6 dd53e2b487da03fd02396306d248cda0
e99f33420f577ee8ce54b67080a80d1e c69821bcb6a8839396f9652b6ff72a70

d131dd02c5e6eec4693d9a0698aff95c 2fcab50712467eab4004583eb8fb7f89
55ad340609f4b30283e4888325f1415a 085125e8f7cdc99fd91dbd7280373c5b
d8823e3156348f5bae6dacd436c919c6 dd53e23487da03fd02396306d248cda0
e99f33420f577ee8ce54b67080280d1e c69821bcb6a8839396f965ab6ff72a70
```

Figure 3-15 Two MD5 message blocks hashed to the same value [19]

As discussed in Chapter 2, another application of cryptographic hashing algorithms for information security applications is digital signatures. Like passwords, the security of digital signatures depends on the cryptographic strength of the hash function implemented [19]. A common example would be signing documents using MD5 to verify the data integrity when being sent over the Internet or a network. In a hypothetical scenario: Two documents are represented by the message blocks above, the first block is a valid document, and the second is malicious. If a person were to sign the valid document indicating the validity of sensitive information with an MD5 hash, they also unintentionally signed the malicious document as both hash to the same value. In the event an untrusted source intercepted the valid document through transmission, they could send their malicious document to the receiver who would believe the document is legitimate as the expected signature is valid. Previous work on MD5 collisions between 2004 and 2007, showed that the use of this hash function in digital signatures can lead to theoretical attack scenarios [20].

Seeing MD5 does not satisfy the fourth requirement of cryptographic hashing algorithms being collision resistant, it can be said the algorithm is cryptographically broken in this regard. Referring to the other conditions, MD5 remains to satisfy them as no evidence or information proving otherwise is present. Thus, producing collisions is the only primary weakness of the algorithm cryptographically speaking and should not be implemented for information security applications involving digital signatures due to this.

In regards to the applicability of MD5 applied for password hashing, there have been no disclosed collisions for a specified range of possible passwords. While it is theoretically possible for a collision to exist, it would be extremely infeasible due to time and memory constraints to determine such a case as the set of possible passwords in a given range would need to be stored then matched against all other possibilities. With this said MD5's applicability to password hashing must be measured based on comparison to other cryptographic hashing algorithms with respect to how effectively each function protects a plain text. This comparison will be evaluated based on the data from the user password complexity studies, along with the results the algorithms produce when their hash values are attacked by various methods.

3.2.4 Secure Hash Algorithm 1

In 1995, a hashing algorithm known as secure hash algorithm 1 (SHA-1) based on SHA-0, was created by the National Security Agency (NSA), and published by the National Institute of Standards and Technology (NIST) as a Federal Processing Standard. SHA-1 is part of the SHA family (SHA-0, SHA-1, SHA-2, SHA-3) created by the NSA and implemented across many government platforms after its publication. The output of SHA-1 is a 160-bit (20-byte) hash value accepting messages less than 2^{64} bits as input. The hash value is typically rendered as a hexadecimal number, 40 digits long. The design of SHA-1 was based on MD4 and MD5's design principles. In terms of operations used in the function, they are the same as MD5 [33] [34]. As with the message digest family, new versions of SHA are created to replace its predecessor's weaknesses. The SHA-2 family, for example, was developed in 2001 as the realization by NIST that SHA-1's output length of 160 bits was too short to justify use over the next decade. As of today, SHA-2 has withstood cryptanalysis and is still relatively safe to use in applications [20].

The SHA-2 family is comprised of six hash functions being SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, and SHA-512/256 where the numbers represent the output length in bits. Differentiation of the functions is truncated versions of one another, used for optimization purposes across different applications for implementation. Another difference is simply the output length of the hash value, which is presented in Table 3-8. The most well known of the SHA-2 family are SHA-256 and SHA-512.

Cryptographic Hashing Algorithm	**Output Hash Value of "password"**
MD5	5f4dcc3b5aa765d61d8327deb882cf99
SHA-1	5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8
SHA-256	5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8
SHA-512	5139b82e47847baf441c245a136b2fabe64f63be0068bce36c5c736269f119873dc6b56b83e5e59cd736768d62a61e57390b16f18fa72c9f8b9842bc2c3b77e9

Table 3-8 Hash outputs from different algorithms for the string password

After comparing the differences of the hash values in the table above, an obvious conclusion can be established. An output of a greater bit length will require more storage and time to compute the values. While this is true, the performance difference in terms of speed between SHA-1, SHA-256, and SHA-512 only differs slightly as one algorithm does not stand out from the rest. Non-conventionally speaking, a cryptographic hashing algorithm that is capable of deliberately producing results slowly is more secure in comparison to very fast algorithms. The validity of this statement is shown in section 3.3.

In Table 3-9 below, the comparison of the performance of the various algorithms running on an Intel Core 2, 1.83 GHz processor running under Windows Vista 32-bit mode is shown [46]. With the hardware available today, the speeds in this table can easily be up to three times as fast. It is important to keep in mind MD5 is capable of producing over 100 megabytes (MB) more data than the other algorithms. On a large scale of data generation, it will be seen in section 3.3 just how much data MD5 and SHA-1 are capable of generating a second, which will be used to prove ultimately how inadequate the algorithm is for security purposes today.

	MiB/Second	**MB/Second**
MD5	255	267.387
SHA-1	153	160.432
SHA-256	111	116.392
SHA-512	99	103.809

Table 3-9 Comparison of data generation between hashing algorithms

Known Weaknesses of SHA-1

Unlike MD5, SHA-1 as of today has no published collisions with message blocks or passwords. On the other hand, NIST disclaimed the use of SHA-1 as of 2005; weaknesses in the function's design were detected by cryptanalysts indicating collisions could be found with fewer required computations that that of a brute force attack. For this reason, NIST required the replacement of SHA-1 to SHA-2 by 2010. Most recently, "Google and Mozilla announced their respective browsers will no longer trust encrypted SSL certificates with expiration dates past December 31st, 2016" [63]. On the contrary, the replacement of SHA-1 has not made significant progress in comparison to MD5, the replacement from common applications is still a large work in progress [20] [35]. The indication of the algorithms popularity can be seen in Appendix A.

Later in 2005, after the discovery of SHA-1's weaknesses, an attack carried out by Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu presented results requiring fewer than 2^{69} operations to find a collision in SHA-1. In comparison to a brute force search, the number of operations required would be 2^{80} [34].

Most recently, in 2015, a group of individuals by the names Marc Stevens, Pierre Karpman, and Thomas Peyrin tested an attack designed by Marc Steven to generate a SHA-1 collision. Marc Steven has claimed to find near working collisions against SHA-1 with $2^{57.5}$ operations in 2010. A freestart collision ended up being found using a 16-node cluster with 64 Graphical Processing Units (GPUs). The authors noted a true collision could be found for the cost of 75,000$ - 120,000$ US, which is within the budget of a criminal organization and the NSA [36] [38] [39].

Overall, SHA-1 has shown to be less resilient to collisions than anticipated and can be harmful to applications, which employ the algorithm for the use of digital signatures. It is recommended, protocols, which use digital signature algorithms should move away from SHA-1 and use an algorithm with greater collision resistance such as SHA-2 [37]. In addition to MD5, the same measures to determine how effective SHA-1 is in respect to safely storing passwords will be used for comparison. Today, with this published information on MD5 and SHA-1, the algorithms are still the most commonly used even with the knowledge of MD5's security vulnerabilities and SHA-1 becoming weaker over time. As of currently, the replacement of MD5 to another more secure cryptographic hashing function is far from completion let alone SHA-1, which has barely started [20].

3.2.5 Conclusions on Cryptographic Hashing

Inclosure of this section, the information covered was an introduction and definition of cryptographic hashing, the applications of the algorithms, their associated requirements, and current research on the weaknesses of MD5 and SHA-1. The primary weakness of the algorithms being failure to satisfy collision resistance is detrimental and can lead to attack scenarios with the application of digital signatures. In regards to passwords, known research on the other requirements being proved false has not been shown, nor has there been a collision for a given range of potential passwords. In order to prove these algorithms are not suitable for password storage, a comparison based on their performance in attack scenarios will be discussed and analyzed in the next section.

3.3 Methods to Compromise Cryptographic Hashes

As the currently known weakness of MD5 and SHA-1 were discussed previously, this section will aim to illustrate how potent and ineffective these functions are when faced with attack scenarios for securely protecting a user. Methods used to recover plain text from hash values and their adequacy with respect to MD5 and SHA-1 are covered. Four simple methods for finding the plain text of password hashes, namely brute force, dictionary attack, lookup tables, and reverse lookup tables in addition to more advanced techniques such as rainbow tables and advanced dictionary attacks will be analyzed.

3.3.1 Resources

At the most fundamental level, the degree of complexity associated with determining a password revolves around the duration of time required to crack the hash value successfully. To achieve such a goal, the resources necessary would be hardware, software and/or any background information such as possessing the hash values themselves, and the security parameters implemented on the developer’s end [24]. Thus, more sophisticated technology and qualitative resources at the disposal of a potential attacker will result in a more efficient approach to password cracking.

Only two types of approaches exist in regards to password cracking: online and offline attacks. Online attacks simply states the attacker will take a potential username and attempt to log-in to the system when it is currently live by any means necessary. An offline attack is a scenario when an attacker has acquired a copy of a system’s database contents and proceeds to employ attack techniques on the password file/table [24]. The primary difference between the approaches is a limited number of attempts and time (assuming the system is setup correctly) in comparison to an unlimited number of attempts and time, respectively.

3.3.2 Brute Force Attack

One of the most fundamental, yet common attacks when cracking passwords is the brute force attack. Brute force does not entail of any elaborate complex algorithms or techniques to crack passwords; rather, it takes the approach of trying every possible combination of characters until the desired result is reached. Given a hash value of a four-character password the brute force implementation would start at “aaaa” then increment by one letter each iteration until the password’s hash is reached, revealing the plain text. Specifically, brute forcing a password should only be applied in offline attacks, as a typical online system would not allow an unlimited number of password attempts for a given user account.

Despite all other techniques, a brute force attack will always crack a password; it is just a matter of how long it will take. Three primary questions need to be asked before considering this type of attack: how much time is needed, how many possibilities can be tested every second, and how many combinations are there [24]. The answers to these questions will depend heavily on whether or not this approach should be considered. With this said, the more powerful an attacker's computer or the number of resources he or she has available, the easier brute forcing a password will be for them. Thus, the trade-off of this technique is a guaranteed result, at the duration of time required to acquire the outcome.

Brute Force Results on MD5

Based on MD5, the effectiveness of a brute force attack will be analyzed in unique scenarios where two theoretical attackers have a different number of resources available. Specifically, the resources used are graphical processing units (GPUs) where the number of MD5 hashes generated a second, will be analyzed. The first attacker's resources are two ATI Radeon 7979 cards while the second attacker's resources are five servers equipped with 25 AMD Radeon GPU's communicating over InfiniBand switched fabric [24]. The results are as follows in Table 3-10. As the results were acquired from two difference sources, the choice of algorithms differed. The exact match-up of algorithms is not present because of this. The most prominent factor is the comparison between MD5 and SHA-1 for this literature. According to these results, the first attacker would be able to brute force:

- All six character passwords in 47 seconds.
- All seven character passwords in 74 minutes.
- All eight character passwords in 465 days.

Algorithm	Tries/Second	Algorithm	Tries/Second
MD5	23,070.7 M/s	MD5	180,000 M/s
SHA-1	7973.8 M/s	SHA-1	63,000 M/s
SHA-256	3110.2 M/s	LM	20,000 M/s
SHA-512	267.1 M/s	SHA-512crypt	364,000
NTLM	44,035.3 M/s	NTLM	348,000 M/s
DES	185.1 M/s	Bcrypt(05)	71,000

Attacker 1 Results [25] *Table 3-10 Hashes per second results between two attackers* *Attacker 2 Results [24]*

Referring back to the statistics on password length in section 3.1.5 for rockyou.com, attacker one would be able to crack all passwords less than eight characters being 33.00% of users slightly over an hour. Since 26.00% of the users of the website used only lower alpha passwords and 2.76% used only mixed alphanumeric passwords, the results differ tremendously if attacker one only covered this character set on the 28.76% of users.

- All six character passwords in 3 seconds.
- All seven character passwords in 4 minutes.
- All eight character passwords in 4 hours.
- All nine character passwords in 10 days.
- All ten character passwords in 625 days.

Attacker two's results in comparison would be dramatically faster; with 7.8 times (based off of the number of MD5 operations per second) the computational power of attacker one, all eight character passwords containing all character types could be cracked in 59.6 days. In addition, all ten-character passwords consisting of only mixed alphanumeric could be iterated through in 80.1 days. It is worth noting the significant differences between the slower hashing algorithms in the results, namely Bcrypt (discussed in section 3.4).

Brute Force Results on SHA-1

Regarding effectiveness, a brute force algorithm applied to SHA-1 has the following results based on all the criteria in the previous section. The data for attacker one is as follows:

- All six character passwords in 135 seconds.
- All seven character passwords in a little over three hours.
- All eight character passwords in 3.6 years.
- All six character passwords in 8.67 seconds (Upper/Lower/Digits).
- All seven character passwords in 11.56 minutes (Upper/Lower/Digits).
- All eight character passwords in 11.56 hours (Upper/Lower/Digits).
- All nine character passwords in 28.9 days (Upper/Lower/Digits).
- All ten character passwords in 5 years (Upper/Lower/Digits).

With attacker two's computational power being 7.8 times greater, they could crack all eight-character passwords of every character set for SHA-1 in 168.4 days and all ten character passwords consisting of only mixed alphanumeric in 233.97 days. This is one example, which illustrates the relation between the implemented hashing algorithm and the complexity of the user's password.

It is quite evident the users with passwords of the greatest length will be targeted last chronologically in a brute force attack, as every password of a length below will be revealed first. With the comparison of time to crack all passwords of length eight involving every character set being 59.6 days to 168.4 days with MD5 and SHA-1 respectively, the latter choice clearly offers more protection due to slower hashing. If a user were to create a password of length eleven they would be out of the scope for being compromised for at least a decade against a brute force attack in this scenario.

3.3.3 Dictionary Attacks

With the knowledge of the lengthy time constraints affiliated with a brute force attack in certain circumstances, a less time consuming yet effective method to apply is a dictionary attack. The attacker first gathers the passwords with the highest potential in a list (the dictionary) then applies an implementation to test them one by one [24]. The dictionary could consist of a large variety of possible passwords including but not limited to the top passwords found by various Google searches, multiple dictionaries such as Wikipedia's, or any potential passwords the attacker might add as candidates.

Typically, prior to a brute force attack, an attacker will attempt a dictionary attack as the number of possibilities in comparison to the largest dictionaries will still be less than that of a brute force approach. In accordance with the types of approach, a dictionary attack will have a greater degree of success in an offline attack, which is under the assumption a system is setup to send security alerts after an extended number of attempts on a given account or from a specific IP address [24].

The primary advantage of this type of attack is the low duration of time it needs to complete. In comparison to a standard brute force attack, which focuses on searching a large proportion of the keyspace, the dictionary attack only attempts the possibilities most likely to succeed. Utilizing this attack generally results in success due to the tendency of users choosing short passwords [40]. Again, referring back to the user password complexity study, this statement yields to be true.

Advanced Dictionary Attacks

The area where simple dictionary attacks fail to succeed is when users formulate passwords based on actual words with a syntax that is unorthodox to the word itself. This usually results when users replace specific characters of words with substitutions creating an identical word with letters, numbers, and symbols. Substituting letters in this manner is very popular amongst users and has its own name known as ("leet speak"). An example would be replacing the letter 'a' with a "@" sign or the letter 'o' with a '0'. Using the word "password" the leet speak representation would be "p@ssw0rd." From a user's perspective, this appears to be a clever technique to keep memorable passwords with the addition of strengthening them. On the other hand, an attacker can easily compensate by using an advanced dictionary attack. This attack lies between a simple dictionary attack and a brute force attack because an attacker would iterate through the entire original word list, replacing letters of each word with leet representations then using the modified word lists in addition to the original word list during an attack [24]. Analysis of password habits and a comprehensive understanding of user password generating habits benefit attackers greatly as they can easily enhance basic attack methods.

Lookup Tables and Reverse Lookup Tables

In computer science, a lookup table, as the name indicates, is used to look up particular items in a table data structure without searching for the item by traditional means; rather, searching is accomplished through an efficient indexing scheme. All elements in the table itself are indexed, allowing a rapid lookup of an item by searching for its key value, where the corresponding element location in the table is found [23]. The indexing scheme to map elements to indices is accomplished through a hashing function as explained in Chapter 2.

Lookup tables are a common and very effective means to attack hashes as they are generated by pre-computing password hashes, then storing the mapping between the password plain text and its corresponding hash value. Commonly, an attacker will only create lookup tables based on a password dictionary as memory constraints become an issue when storing billions of password/hash value pairs. Based on the results from section 3.1.5 on user password complexity, creating lookup tables based on words would achieve the highest results. An example lookup table is displayed in Table 3-11. To use a table such as the one below a person would simply take any hash from a compromised user's account, and then search through the lookup table for the corresponding plain text password. Even if a lookup table contains billions of hash values, hundreds of hash values can be looked up each second, assuming the lookup table has a optimal implementation [22].

Passwords	**Hash Values**
Password	5f4dcc3b5aa765d61d8327deb882cf99
123456	e10adc3949ba59abbe56e057f20f883e
12345678	25d55ad283aa400af464c76d713c07ad
Qwerty	d8578edf8458ce06fbc5bb76a58c5ca4
Dragon	8621ffdbc5698829397d97767ac13db3

Table 3-11 A Lookup table

While lookup tables are very efficient in the mentioned manner, they are not applicable for storing plain text and hash value pairs of a given character set. If a person wanted to accomplish this, they would need to store every possible combination of characters for a particular length. This would cost an unreasonable amount of memory past six character passwords. From a firsthand calculation, six terabytes of memory would be required to store every possible combination of strictly lower alpha for a string of length eight. This excludes the hash value. To store 95^8 potential passwords with their associated hash value would be infeasible and cumbersome in terms of resources required and time.

Reverse lookup tables, as the name suggests, is used to perform the opposite task of a normal lookup table. Instead of storing the plain text and hash value pairs, an attacker creates a normal lookup table of the compromised user account names and hash values, and then takes identical hash values common between unique accounts. These unique account names are then grouped with the knowledge all the accounts have the same password. No pre-computation is required when attacking the password file; instead, a password guess is generated along with its hash value, which is then compared to the hash values in the lookup table. If a match is found all the users with, the corresponding hash values are found, thus, allowing multiple users to be attacked simultaneously. In the lookup table in Table 3-12, if the password guess matched one of those hash values, the plain text password for all the associated users would be known.

Username	**Hash Values**
Bob, Alice	5e8f9593febf28839beb29c1cb84f182
Mike, Joe, Rick	fe2d5a2678b61a6dc1e113ddcc073bcf
John	25d55ad283aa400af464c76d713c07ad

Table 3-12 A reverse Lookup table

Dictionary Attacks on Hashing Algorithms

With the nature of dictionary attacks being focused on candidate passwords with the highest potential to be cracked, the cryptographic hashing algorithm implemented has only a slight significance on the effectiveness of such an attack. The impact the chosen algorithm would inflict is the duration of time needed for completion of pre-processing and the overall performance in regards to search speed. Referring back to Table 3-9 from section 3.2, the data generation of the algorithms is shown once again for reference below in Table 3-13.

Since dictionary attacks typically cover the set of words in a given dictionary, in addition to the most frequently used passwords by users, the number of iterations required for searching is far less than that of a brute force attack. This is especially true with a binary search implementation. An estimate of a large dictionary attack would be upper bounds of 10 billion possible words. Even then, the duration of time to complete the attack with all four algorithms shown below in the chart, on a high-end system would be less than an hour (depending on the hardware used). Since the time required to complete this type of attack is not an issue, the success rate revolves heavily around the user choosing a password present in the lookup table or based on words. In this case, no hashing algorithm will effectively protect a user. An explicit password enforcement scheme is needed otherwise, as mentioned in section 3.4.

	MiB/Second	**MB/Second**
MD5	255	267.387
SHA-1	153	160.432
SHA-256	111	116.392
SHA-512	99	103.809

Table 3-13 Comparison of data generation between hashing algorithms

A study performed by Joseph Bonneau in 2012, with the consent of Yahoo, analyzed 70 million passwords of the website's users from dozens of subpopulation using different hacking attempts to develop a statistical analysis on the passwords. Bonneau's dictionary attack parameter was using only 1000 of the most common words in the dictionary with a specialized algorithm to determine the plain text of the passwords. The algorithm was able to guess correctly 10% of the user's passwords based on his chosen list. Across the multiple subpopulations, the password creation schemes were similar in nature without a deviance of a certain group creating a significant number of strong passwords. Bonneau concluded the weak passwords are an underlying problem as users are not able to manage the degree of difficulty associated with guessing their weak passwords [42].

For firsthand experimentation on the effectiveness of lookup tables with a dictionary attack, the website Crack Station[3] claims to contain every wordlist, dictionary, and password database leak the author of the website could find on the Internet, in addition to every word in the Wikipedia database and password lists from various database breaches. Applying different MD5 and SHA-1 hashes (based strictly on English word combinations) the online application was able to determine fifteen character passwords in less than a few seconds.

3.3.4 Rainbow Tables

"A Cryptanalytic Time – Memory Trade-Off" was a paper published by Martin Hellman in 1980, discussing the precomputation of cryptologic functions with the intent of using the stored values to quickly look up others [67]. This technique became known as "rainbow tables" after effectiveness with lists of precomputed hashes to find others was successful [69].

Unlike lookup tables, which store the hashes of each generated plain text in memory, rainbow tables store chains of hashes, which will be discussed shortly. As the main problem of lookup tables is memory, rainbow tables resolve this issue as they are a compromise between pre-computation and low memory usage [8] [45]. In order to generate such a table two main components being a reduction function and a hash chain need to be implemented.

Essentially rainbow tables are a pre-computation based approach to reversing hashes. Much pre-computation is required to generate the tables, but the benefit is quite significant. The effectiveness is achieved post pre-computation. Searching for hashes throughout the tables requires significantly

[3] Crack Station: https://crackstation.net

less computation than that necessary for the pre-computation of the tables and that of brute forcing a password [9].

Reduction Functions

The first component being reduction functions is the backbone of rainbow tables. In order to fully understand rainbow tables, one must comprehend the workings of the reduction function. As discussed earlier, a cryptographic hash function takes a given plain text and transforms it into a hash value where the result is a unique mapping representation between the plain text and the hash value. Reduction functions, on the other hand, can be thought to perform the opposite function as that of a cryptographic hashing function. Instead of mapping a plain text to a hash value, the reduction function uniquely maps a hash value to a plain text. In terms of similarity between the functions both are one way. Figure 3-16 illustrates a starting plain text, which is first hashed, then reduced. It is worth noting a reduction function does not reverse the given hash back to its plain text; it is not an inverse hash function. As mentioned previously a cryptographic hash function's purpose is to generate a mapping between a plain text and a hash with the criteria of an inverse function being impossible to create. For example, take the hash of a given plain text, then use that hash as input to a given reduction function. The original plain text will not be the output, but rather some other plain text. It is important to note the output plain text is dependent on the parameters assigned to the reduction function; it is not random, but rather predefined [8] [45].

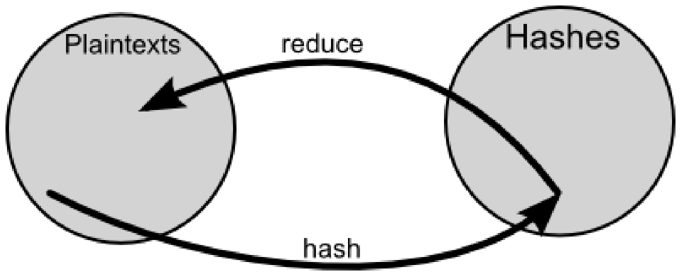


Figure 3-16 The high level process of going from a plain text to a hash value back to a different plain text

For instance, if the set of plaintexts for a given password is [abcde]{4}, where four represents the maximum length of a string derived from the set, to generate a rainbow table of all passwords in this set, the reduction function R () could have its parameter set to extract the first four characters of a given hash. If the input to the reduction function is, the MD5 hash of the plain text "pass" the output of the function would be "fdfa." Thus, we have produced a different plain text from the hash of the original plain text, which is the purpose of a reduction function. To generate the rest of the possibilities in this set, the result from the reduction function would be hashed, and then used as input to the reduction function once again. The process ends once the function has iterated ‘x’ number of times, where ‘x’ represents the number of possible passwords in the set. The following shows a basic example of the process.

MD5 (“pass”) → “966f77d7531f7a266beba3bada3894e6”

R (“966f77d7531f7a266beba3bada3894e6”) → “fdfa”

Hash Chains

The second component of rainbow tables being the hash chains is a direct result from multiple iterations of the reduction function. A link in a hash chain is simply a plain text hash value pair. In the example above, one link would be the pair: (pass, 966f77d7531f7a266beba3bada3894e6). A subsequent link would be “fdfa” and its hash value. Through multiple iterations of this process, a hash chain can be generated. Every chain will always start at an arbitrary plain text and end with the hash value of the most previously generated plain text. Consequently, this only requires the starting plain text and ending hash value of a chain to be stored in memory. Through this, the pairs between the starting plain text and the ending hash can be stored abstractly with low memory usage. This factor is what allows rainbow tables to be very memory efficient as billions of hashes can be stored abstractly. Depending on the criteria of the rainbow table being generated, the number of pairs can easily be in the millions and upwards as the amount of memory for storing two strings is negligible [8] [45].

In regards to the length of a hash chain, any number is viable, but typically, rainbow tables have hash chains of 1000 links upwards into the millions. Different hash chains can have a variance in length as well. The programmer could explicitly decide when a new chain would be started based on the rainbow table they are generating. For the reason of perspective regarding the size of rainbow tables, a typical table can be easily up to 300 GB covering tens of billions of different plain text hash value pairs. When generating a rainbow table, the goal is to try to cover 99.9% of the set for a given range of possible passwords. For illustration purposes, a rainbow table with its associated hash chains is displayed in Figure 3-17. It is worth noting rainbow tables get their name from the use of a different reduction function being used in each chain where if the table entries were colored they would look like a long vertical stretch of a rainbow.

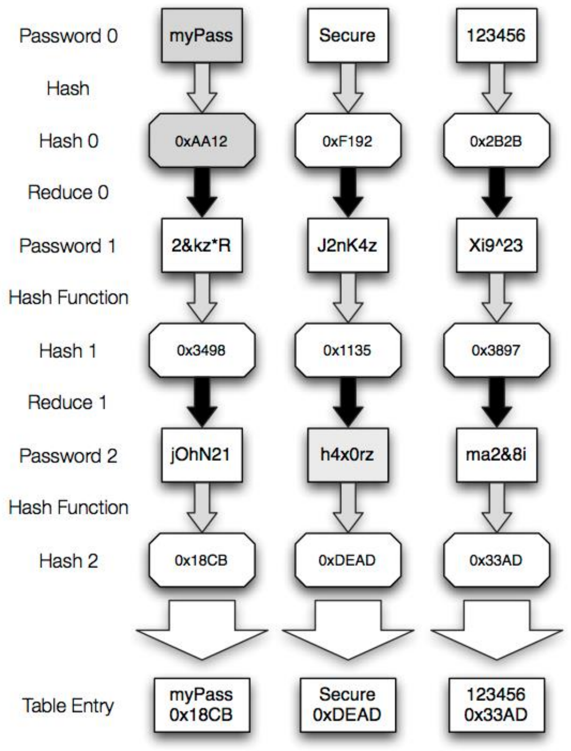


Figure 3-17 Hash chains

Rainbow tables are a tradeoff between time and memory. Specifically, when generating a rainbow table, an individual can specify a small set of chains with a large number of links. The result is a minimal amount of required memory, with the trade-off of additional computational time to search for a hash value. The other alternative is generating a large number of chains with a shorter number of links. This results in much more memory, but less time to search for a hash value. In both scenario's ramifications, rainbow tables still require less time than a brute force attack when dealing with passwords past the length of six characters.

Determining the Plain text from a Hash Value

Once a rainbow table has finished generating, the table can be used to determine an unknown plain text from a given hash value. Although this is under the assumption, the plain text for the given hash exists within a chain. To ascertain the plain text of a hash value from the hash chains, the following algorithm is used [8] [45].

1) Look for the given hash value in the list of final hashes, if the hash exists in one of the chains break out of the loop.
2) If the given hash does not exist, reduce the given hash into another plain text, and hash the new plain text. Once the number of reductions exceeds the length of the static chain lengths in the table, exit the loop as the value is not in the table.
3) Go to step 1.
4) If the given hash matches one of the final hashes, the chain for which the hash matches the final hash contains the original hash.

After a match is made, the starting plain text of that hash chain is retrieved. From here on, the specific chain is reproduced until the given hash is found in the chain while iterating. Once the given hash is found in the chain, its corresponding plain text is the previously generated plain text. For example, if the unknown plain text of the hash 0x1135 (shown in Figure 3-18, chain 2) is desired, the hash would be compared to all the end chain hash values being 0x18CB, 0xDEAD, and 0x33AD respectively. In this case, since none of the hashes match, 0x1135 would be reduced to the plain text "h4x0rz," which would then be hashed to 0xDEAD. Through searching the end hash values of the chains once again for the newly generated hash value, a match is found. The chain's starting plain text is obtained; being "Secure," then would be hashed, and then reduced until the original hash value is found. Once 0x1135 is found the resulting plain text of this hash would be "J2nK4z" thus being the previous plain text.

Problems with Rainbow Tables

As rainbow tables are a significantly more effective approach compared to other password cracking methods, a few primary issues exist. To achieve results from rainbow tables, a person needs to be aware of when they should or should not use them based on the security measures implemented on hash values. Specifically, these issues are password salts, collisions, and guaranteed results.

As discussed previously, a password salt is used to randomize the hashes to secure them further with the intention to achieve a greater number of unique hash values stored in the database. Since the result of salting a password has been generating a different hash value for the same password, rainbow tables become ineffective. Rainbow tables exploit the fact password hashes are generated the same way, thus salting passwords detriment rainbow tables. The simple reasoning is because the salt is not known in advance and since rainbow tables are pre-computed, the attacker will be in the absence of a rainbow table with the used salt. Following is a scenario that explains this situation.

An attacker has a rainbow table for eight character passwords and obtains a database with millions of hashes, which are determined to be salted with a ten character salt value, all being unique. Any user password being eight characters long now becomes 18 characters after the salt is applied. With the addition of ten characters added to the original password, the eight character rainbow tables become ineffective. To compensate for this, the attacker would need to regenerate a rainbow table for every single salt used for each password. This severely inconveniences the attacker as to crack one password they would need to generate an entire rainbow table. It is worth mentioning the longer the salt value, the more time is required to generate a rainbow table or lookup table. If the salt was only three characters long, an attacker could easily generate a lookup table for every possible salt.

Collisions discussed in section 3.2 are the second problem for rainbow tables as ironically an algorithm which generates collisions becomes more secure in regards to rainbow tables that is. In the event multiple plaintexts map to a single hash value, the hash chains of a rainbow table will eventually converge into one. This can be seen in Figure 3-18. In addition, loops become another issue. If two plaintexts in a chain are hashed to the same value a loop will occur where the same plain text-hash value pair are generated over and over again until the chain ends [8] [45].

Lastly, unlike a brute force algorithm, there is no guarantee a rainbow table will find a hash match in one of its chains. This means a password will only be located if it exists in the search space. The algorithm is problematic when it comes to coverage of all possibilities, as it is very difficult to ensure all passwords have been generated. However, very high success probabilities can be achieved, and the search time is significantly less than a brute force algorithm [9].

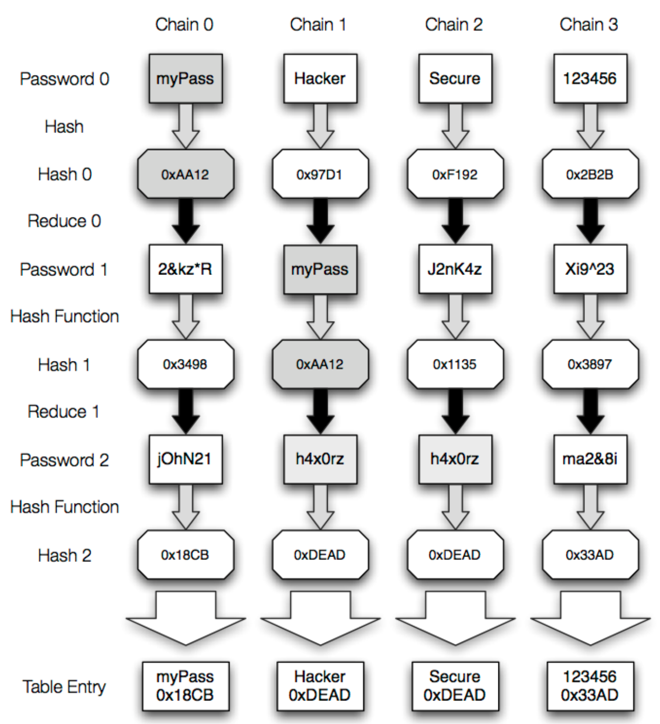


Figure 3-18 Collision in a rainbow table

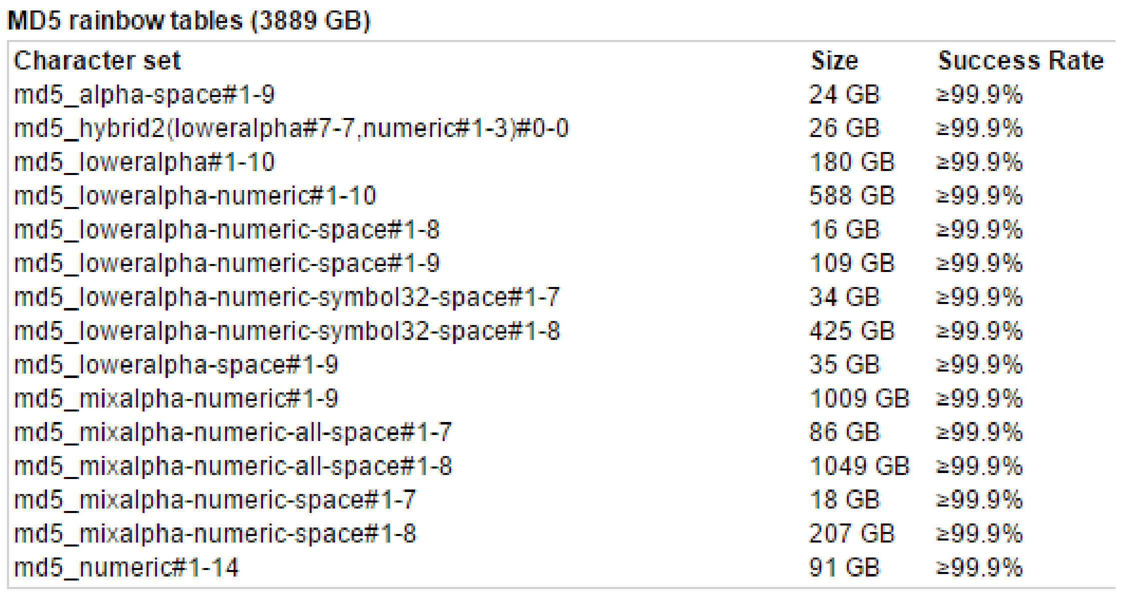
Rainbow Table Effectiveness

As mentioned, rainbow tables are not guaranteed to obtain a password based on its search space. Either it is present in one of the table's various chains or not. Although, with this said very high probabilities of success are achievable with the complement of the search time being significantly less than that of a brute force attack. Often, the percentage of a successful rainbow table attack can reach 99.9% or higher optimally [45]. Primarily, the effectiveness of rainbow tables is based on a few factors:

- The hashes must be unsalted. Otherwise, rainbow tables become futile.
- The success rate percentage, which should be typically 99.9%.
- Whether or not the password lies in the keyspace.

As there is an absence of reports for rainbow tables indicating their successful results in cracking hashes, the analysis of effectiveness with this attack method will have to be carried out based on generation of rainbow tables first hand. Alternatively, and individual could use the websites Project

Rainbow Crack[4] and Free Rainbow Tables[5] to download rainbow tables. Each site contains over 10 TB of pre-computed data covering various hashing algorithms. The largest amounts of data are associated with MD5 and SHA-1 as seen in the tables 3-14 and 3-15 (Appendix G).



MD5 rainbow tables (3889 GB)

Character set	Size	Success Rate
md5_alpha-space#1-9	24 GB	≥99.9%
md5_hybrid2(loweralpha#7-7,numeric#1-3)#0-0	26 GB	≥99.9%
md5_loweralpha#1-10	180 GB	≥99.9%
md5_loweralpha-numeric#1-10	588 GB	≥99.9%
md5_loweralpha-numeric-space#1-8	16 GB	≥99.9%
md5_loweralpha-numeric-space#1-9	109 GB	≥99.9%
md5_loweralpha-numeric-symbol32-space#1-7	34 GB	≥99.9%
md5_loweralpha-numeric-symbol32-space#1-8	425 GB	≥99.9%
md5_loweralpha-space#1-9	35 GB	≥99.9%
md5_mixalpha-numeric#1-9	1009 GB	≥99.9%
md5_mixalpha-numeric-all-space#1-7	86 GB	≥99.9%
md5_mixalpha-numeric-all-space#1-8	1049 GB	≥99.9%
md5_mixalpha-numeric-space#1-7	18 GB	≥99.9%
md5_mixalpha-numeric-space#1-8	207 GB	≥99.9%
md5_numeric#1-14	91 GB	≥99.9%

Table 3-14 freerainbowtables.com MD5 Rainbow tables for download [70]

In the Table 3-14 and Appendix G, each row represents a different rainbow table covering a diverse range of password lengths represented by the digits after the '#'. A focal point is some of the tables cover up to ten character passwords. The second website illustrates proof of the success, as there are three links to YouTube videos where two of the links are for MD5 and SHA-1 hash values. The videos demonstrate a program utilizing the rainbow tables to determine the plain text of a given hash value. As far as the results of the videos are concerned, in nine minutes, an eight-character MD5 password covering all 95 possible characters was cracked. In eleven minutes a different eight-character SHA-1 password with the same character set was revealed as well.

Due to a lack of available statistics on the successes and effectiveness of rainbow tables, Chapter 5 will present the first hand experimentation of utilizing a special piece of software known as "rainbowcrack" to generate then search through the tables. Furthermore, results and analyzation on the experiment are discussed afterwards.

[4] http://project-rainbowcrack.com/
[5] https://www.freerainbowtables.com/

3.3.5 Conclusions of the Attack Methods

In conclusion of this section, multiple unique attacks commonly used by potential attackers were discussed, namely, brute force, dictionary, advanced dictionary, lookup tables, reverse lookup tables, and finally, rainbow table attacks. Based on these overall results, it has become clear MD5, and SHA-1 are not acceptable for the application of protecting passwords, not because of their cryptographic weakness, but rather due to their outstanding speeds at generating hash value data. The throughput these algorithms are capable of producing is beneficial to optimize server performance, although, the tradeoff is providing optimization to potential attacks on innovative technology. With hardware becoming cheaper and faster, the algorithms will only continue to deteriorate as the processing of billions of hash values per second will only increase.

In parallel, since users maintain the same weak password schemes as time goes on, the cracking of password hashes will only become easier for attackers. In addition, the vast amount of pre-computed data available to assist attackers such as the copious amount of terabytes available for rainbow tables to be downloaded extends the potency of the algorithms in the event of an attack. The next section will discuss countermeasures, which can be taken to mitigate the mentioned approaches.

3.4 Effectively Securing Passwords

This section presents a known technique to effectually protect against rainbow tables, lookup tables, reverse lookup tables, and dictionary attacks by increasing the duration of time required to perform the attacks making them less effective. Following is an alternative means to store password hashes using key derivation algorithms, which drastically detriment brute force attacks and the other methods discussed. On the developer's end, implementing these alternatives will provide a strong degree of protection from a standard cryptographic hashing algorithm implementation.

3.4.1 Salting and Peppering Passwords

Salting a password, is simply concatenating a random string with a plain text to add a significant degree of complexity to offline attacks such as password guessing [21]. Peppering a password is the same process as salting with the difference of adding the random string in front of the password as opposed to concatenating to the end. The difference can be seen as follows: hash (password +

salt), hash (pepper + password). It can be said salting is appending, and peppering is pre-pending. Unlike the plain text password, the salt does not need to be hashed and can be stored in a separate field, in plain text alongside the hash value in the database.

This is necessary in order to create a mapping between the salt and the password. Salt values can be stored in plain text because an attacker will not know the value in advance. This means they cannot perform pre-computation on the hashes. An example is shown in Table 3-14. Before the entered password is used as input to the cryptographic hashing algorithm, the salt for that particular user is retrieved then augmented to the password itself. Once the augmented password's digest is generated, the comparison between the stored digest can be processed [21]. A high level of this process is illustrated in Figure 3-19.

Username	**Password**	**Salt Value**
Joe	47e6bfad7dca424663c113a23a3cb7e8	8dfj2ndif782hf9s8
Bob	68e36741a1c43857117b943a41381a53	MNJdus8adHDdsf7s2
Alice	fe0cdb0dc63fb53087e4beec1e4eeed4	Jdj3P9d4kjdJd827f

Table 3-16 Illustration representing how a salt value is stored

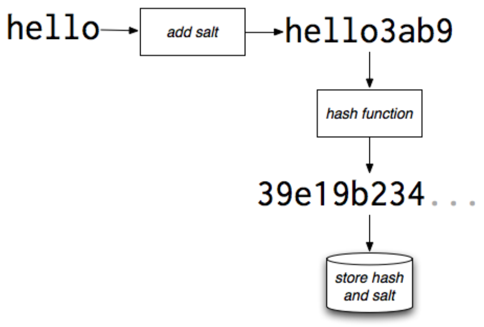


Figure 3-19 A high level illustration of how a password is salted

An important factor to keep in mind about salting and peppering is the process does not increase the complexity of a password itself. In other words, the method does not convert a weak password into a strong one. As an example, if a password is three characters long and the salt is twenty characters, then the entire length of the password is now twenty-three characters. This may seem like the process is strengthening the password, but if an attacker generates a lookup table for all possible three character passwords with the twenty character salt appended, the time needed to obtain that password is not any more significant than the time it would take to reveal the password without the salt. Since the salt is in plain text, the attacker can easily use it to generate a lookup table with the salt value. Therefore, the work factor for revealing the password of a particular user remains unchanged, however, the work factor to generate password hashes, then compare them to the hashes in the database is increased significantly [21].

The primary benefit of adding a salt or pepper is the preemptive outcome to lookup tables, reverse lookup tables, and rainbow tables. All these become ineffective because the salt value is not known in advanced by the attacker. In the circumstance a database is compromised, the attacker would need to generate an entirely new set of tables to attack the password file or resort to a brute force attack [22]. If a random salt is generated for each individual user, an attacker would need to regenerate a separate lookup table or rainbow table for each user as the salt values vary. Using the exact same salt for every user would require an attacker to regenerate only one version of the desired table. Thus, unique salts add an unequivocal degree of additional computation required by an attacker. For this reason, pre-computation becomes ineffective; meaning rainbow tables, reverse lookup tables, and lookup tables are of no use.

In both scenarios of regeneration or brute forcing hash values with salts, the factor of time to reveal the hashes is tremendously increased. It is worth noting salts against brute force attacks become ineffective against well-funded opponents. With a computer cluster of GPU's or servers with the ability to try billions of combinations a second, a salt will not make much of a difference at that point. A second benefit of applying a salt or pepper is an extra layer of security added to passwords, which are common among different users. Since a salt modifies the password before being hashed, identical passwords on different accounts with unique salts will hash to unique values. This means if one account's hash value in a set of many with identical passwords is revealed the passwords of the other accounts will remain masked [21]. An example of this benefit is shown in Table 3-17.

Hash("Plain text + salt")	**Resulting Hash Value (SHA-256)**
Hash("hello")	2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
Hash("hello" + "QxLUF1bgIAdeQX")	9e209040c863f84a31e719795b2577523954739fe5ed3b58a75cff2127075ed1

Hash("hello" + "bv5PeSMfV11Cd")	5142a44801862fd2024bdf6c8fdf0d7fbddab31041 794640ca4bb6f46ae000e5
Hash("hello" + "YYLmfY6IehjZMQ")	a49670c3c18b9e079b9cfaf51634f563dc8ae3070d b2c4a8544305df1b60f007

Table 3-17 Different hash values for the same password with a salt applied [22]

Regarding the length of a salt value, short salts are less effective as the number of possibilities to unmask a hash value is reduced, thus, giving an attacker less time needed to acquire a result. The consensus on this matter is the length of the salt should be the same size as the output of the hash function. In the case of SHA256, which is a 256-bit hash, the salt should be at least 256 bits or 32 bytes [22].

In the event, an attacker decides to utilize a dictionary attack; a salt value will only apply a minimal degree of ineffectiveness to the attack. Since dictionary attacks can be very fast depending on the word count in the list, an attacker would only need to take the salt value and append it to the words in their dictionary. Following, they would then use the result as a parameter to the hashing algorithm. Instead of regenerating the word list, the attacker would only need to compare the new hash value from their word list to the hash value stored in the database. The resulting attack becomes a hybrid between a dictionary and a brute force implementation. In comparison to regenerating large rainbow tables of hundreds of gigabytes worth of data to applying a unique dictionary attack to every individual user, the latter is much more desirable, even with adding 'n' to the number of dictionary attacks required, where n represents the number of compromised users. The time difference between the two differs exponentially.

Using Salts with MD5 and SHA-1

As it has been shown how salts can counteract potential attack methods, MD5 and SHA-1would be improved explicitly with increased resistance to the associated mentioned attacks protected by salts. Different variations of applying the salts are shown below. Again, brute force attacks would not be protected against in this case, if an attacker acquired the specified way the passwords were hashed, they could alter their brute force attack to be optimized with the salt variation. This would mean very little to well-funded and motivated opponents with multiple servers at extensive processing power.

- hash = Hash (password + salt)

- hash = Hash (Hash (password) + salt)
- hash = Hash (Hash(password + salt))

An analysis report on the security of MD5 in 2013 [17] stated the countermeasures such as increased password length, the various ways of salting, improving the overall algorithm, iterative hashing, and key stretching, for example, can be taken to increase resistance or completely counteract the attacks. While all these measures are applicable to SHA-1 as well, if an attacker were to obtain the knowledge of how the algorithm was improved, this would aid them to reverse the measures and ultimately revert back to the problem of determining the MD5 or SHA-1 hash value through brute force or dictionary attacks if necessary. Regardless, while it is true these approaches improve the algorithms; it is aimless and non-pertinent to apply these measures to outdated functions when they could be applied to more secure and modern functions such as the SHA-2 family. With respect to the report, the improvement methods discussed are a great resource to improve password protection further, although to achieve the highest degree of password security a very secure base algorithm for hashing needs to be implemented.

3.4.2 Enforcing Password Requirements

Protecting users on the developer's end should be one of the highest priorities during development. Assuring a user their information is protected and the overall system can be trusted, is a main requirement for applications today. Unfortunately, a developer cannot passively protect their system from dictionary attacks in the event of a breach. Passwords chosen by users considered weak, based on words, are of short length, or have a possibilitly of existing in dictionary lookup tables generated by attackers will be at risk despite the cryptographic hashing algorithm implemented.

As discussed, the cryptographic hashing algorithm will only affect the speed of the attack. In an offline attack, the time difference is meaningless to a motivated attacker. The only method to make dictionary attacks futile is enforcing users to choose strong passwords based on specific requirements during account creation. A developer should take the time to generate his or her own lookup tables of the most common passwords used, and various dictionaries to test if a plain text is vulnerable to dictionary attacks. A comprehensive list of names and locations should also be included in these lists. If a password contains any of these words, a flag should go off notifying the user during account creation. Alternative means to generate passwords such as passphrases discussed in section 3.1 should be mentioned to users. A component, which displays the strength of the password, should be shown to a user as part of the webpage for account creation. Examples of such an interface can be seen in account creation for a Hotmail or Google account. According to Microsoft, the requirements to make a strong password, which should be enforced, are as follows:

- Is, at least, eight characters long.
- Does not contain your username, real name, or company name.
- Does not contain a complete word.
- Is significantly different from previous passwords.
- Contains characters of uppercase letters, lowercase letters, numbers, and symbols [43].

Applying these requirements in addition to the lookup tables to test if a password contains words, a very effective enforcement for users to create strong passwords can be established. Through this, dictionary attacks become ineffective. Being proactive with password security can mean the difference between millions of compromised users to infeasible time constraints for attackers to determine multiple user passwords.

3.4.3 Key Derivation Functions

Unlike the message digest series and the secure hash algorithm family, which generate hash values efficiently and quickly, Key Derivation Functions (KDFs) were designed with the intention of generating output slowly. These algorithms are not classified as cryptographic hashing functions, nor were designed with the intention to be used in such a manner. KDF's are used for encryption purposes, specifically referring to the key value discussed in Chapter 2. In short, a KDF uses the value of a given key as input, and then derives another key based on that input where the new key is of a greater length than that of the original in most cases [28]. This is known as key stretching, which strengthens a given key into a suitable candidate to be used for encrypting data.

The purpose for a KDF is the same of that as weak passwords; weak keys for encryption are vulnerable to brute force attacks. A KDF's input is the initial key, and the output is the enhanced key. Essentially, from a high level perspective, this type of function can be thought of similar to that of a cryptographic hash function. The primary difference is a KDF has specific randomness properties as an algorithmic requirement. The similarity between the two types of algorithms is they are both incredibly difficult to reverse. Thus, valid KDFs can be used for storing passwords as an alternative cryptographic hashing function.

The main benefit of using these types of algorithms as an alternative is the ineffectiveness provided towards brute force attacks. In essence, the functions are capable of deliberately increasing the duration of time required to produce an output by adding a specified number of iterations to the algorithm (typically i^{10}). Through this, the algorithm will keep on performing operations based on the previous output of each iteration until the final output is reached. Not only this, the algorithms require salts to be used as parameters implicitly before execution, adding the security benefits of salts automatically. This means brute force attacks, dictionary attacks, rainbow tables, look-up tables, and reverse lookup tables become near futile. Due to these desirable characteristics, the functions are excellent means for the applications of cryptographic hashing. It is worth noting, if these algorithms were to be implemented, the number of iterations specified as input should be significant to the point where performance on the user's end will not be affected.

Bcrypt and PBKDF2

A key derivation function designed by Niels Provos and David Mazières in 1999 known as Bcrypt is built upon the Blowfish block cipher. Simply said, the Blowfish block cipher allows the algorithm to iterate slowly because it uses a slow key scheduler. For example, Bcrypt running on an AMD HD 7970 graphic card can only produce computations of about 4000 per second [27]. In comparison to the millions of hash values a second that can be generated with MD5 and SHA-1, the effectiveness of the algorithms slow computations can be seen. As input, the algorithm takes a password (key), iteration count (cost), and a salt. Since Bcrypt is classified as KDF, the output value in comparison to MD5, SHA-1, and other cryptographic hashing functions slightly differs. Each Bcrypt hash can be split into four different parts being: the modular crypt format token, the number of key expansions token, the salt value token, and the hash value token. An example of a Bcrypt hash value is broken up in Table 3-18 below.

Bcrypt Output:
"$2a$10$Nnfwu.Qgk2ehByz.rR5vbOMH5vQ8Rtcw8AWPZDDYHzyAzzIyE338W"

Bcrypt Output Parts	Meaning
\$2a\$	This hash string is a Bcrypt hash in modular crypt format.
10\$	2^{10} key expansion rounds (or iterations).
Nnfwu.Qgk2ehByz.rR5vbO	This is the salt value (22 characters base-64 encoded, 128-bit salt).
MH5vQ8Rtcw8AWPZDDYHzyAzzIyE338W	This is the resulting hash (31 characters base-64 bit encoded, 184 bits).

Table 3-18 Associated meanings with the different parts of the output from Bcrypt

To illustrate Bcrypt's effectiveness as a potential candidate algorithm to be implemented in a system for securing passwords, Table 3-10 in section 3.3 displays the number of Bcrypt hash values, which can be generated per second over the five servers. In comparison to MD5 and SHA1, Bcrypt yields a 2,535,211: 1 and 887,323: 1, hashes per second ratio, respectively. Thus, it is evident the effectiveness of an iterative algorithm when faced with a brute force attack.

Password-based key derivation function 2 (PBKDF2) was developed by RSA Laboratories' Public-Key Cryptography Standards (PKCS) series. It is very similar to Bcrypt and is recommended by NIST. Like Bcrypt, the function requires an iteration count (cost), salt, and password (key) as three out of five of its parameters. The other two parameters are a pseudorandom function and the length of the derived key. Through being able to specify the value of key length (hash value), a developer can utilize this to optimize performance on their server(s). In addition, significantly long output values add more protection to the hash values themselves. An example of a PBKDF2 hash value with the assumption of an unknown pseudorandom function can be seen below. The parameters for PBKDF2 are as follows from [47]: DK = PBKDF2 (PRF, P, S, C, dkLen) where:

- PRF is a pseudorandom function of two parameters with output length hLen.(Optional)
- P is the master password from which a derived key is generated.
- S is a sequence of bits, known as a cryptographic salt.
- C is the number of iterations desired.
- dkLen is the desired length of the derived key.
- DK is the generated derived key.

A PBKDF2 example would be DK = PBKDF2 (PRF, “Password”, “xhsud7123”, 1000, 64) where: “Password” is the user’s password, “xhsud7123” is the salt value, 1000 is the number of iterations, and 64 being the number of bytes for the output key. The resulting derived 512-bit key is: “3688b0b6be7a0e6757b3ecbec7ea46bc3f6d0500b617f7e3dd9fdf7f8d21041c6183950b110a3eba 605e7c9236ae9688be9a90e73e1ae9e124aefbb1abfa0c15”.

As these algorithms lack popularity, not because of their credibility to hashing passwords but rather a knowledge aspect, potential attackers tend to overlook these hashes when they can focus their efforts on cracking hashes that are more vulnerable. If a developer wants to implement these algorithms for a system, it is recommended further research should be undertaken. To achieve the added benefit of slow hashing to the standard recommended algorithms such as the SHA-2 family a technique known as iterative hashing can be applied. To apply iterative hashing simply, all that is required is a loop structure around the call to the hash function with the number of desired iterations. An example can be seen below in Java. By adding additional tens of thousands of iterations, a recommended cryptographic hashing algorithm can be slowed significantly.

```
String password = “user-password”; for (int i = 0; i<10000; i++) password = getMD5 (password);
```

3.4.4 Conclusions on Effectively Securing Passwords

Through applying a salt or pepper value, enforcing a password strength scheme in a system, adding additional iterations to a standard cryptographic hashing algorithm, or implementing a KDF can significantly increase the overall security a system provides for both the users and developers. The various mentioned attacks discussed in the previous section can become near futile if proper measures such as the ones presented, are taken to ensure the highest protection for hashes in the back end database. Nearing the end of this chapter, the next section will report additional cases where a system using MD5 or SHA1 was compromised. The results of the attacks prove further the functions inadequacy.

3.5 A Few Results on Attacking Unsalted MD5 and SHA-1 Hashes

Referring back to the Ashley Madison hack, the individual responsible for implementing the password protection scheme decided to use two cryptographic hashing algorithms being Bcrypt and MD5. Assuming MD5 was the first algorithm implemented, the decision of using Bcrypt was a proactive approach to protecting the users. On the other hand, the missing step that led to the website’s security failure was not emailing a password change request to the users whose passwords were hashed with MD5. If the emails were sent, the developers could have required those users to change their passwords in order to eliminate the stored MD5 hash values stored in the database.

Due to this, when the website’s hashes were compromised, 11 million unsalted MD5 hashes ended up being cracked by CynoSure Prime in just ten days. In contrast, when a blogger attempted to crack the Bcrypt hashes, only 4000 were revealed in a week [7]. Those passwords were most likely the weakest. In this case, it is easily seen the security difference between the two algorithms based on an 11 million to 4000 revealed password ratio. The primary advantage of KDF’s is the ability to set the number of iterations; the higher the cost value the most computationally expensive it is to crack the hashes.

Alternative examples of websites using no salting with MD5 and SHA-1 are LinkedIn and eHarmony, respectively. LinkedIn was hacked on June 6th, 2012, utilizing unsalted SHA-1 resulting in 6.5 million hashes uploaded to the Internet. A consultant from Kore Logic Security was quoted to having cracked 55% of the 6.5 million in 24 hours. E-Harmony was hacked on June 5th, 2012 implementing unsalted MD5 where out of the 1.5 million hashes uploaded, 80% were cracked by Trustwave's SpiderLabs Blog in a reported 72 hours. Another case of SHA-1 with no salt values would be an Australian Broadcasting Company. Out of 50,000 hashes, 53% were obtained in 45 seconds by Troy Hunt [44].

In a summary of over 40 compromised databases written in a report by Dennis Mirante and Justin Cappos [44] of the polytechnic institute of NYU, the minimal accumulated information on only 26.5% of the databases was obtained. “Out of this percentage 11.8% of the websites reported their passwords were hashed and unsalted, 5.9% used salted MD5, 14.7% used unsalted MD5, 11.8% used salted SHA-1, while unsalted SHA-1, SHA-256 salted, crypt(3) salted, and Bcrypt each accounted for 2.9%. Plain text use was noted in 17.6% of the site breaches [44]”. Details on the list of hacked websites can be seen at [44].

From this information, it is clear only a majority of websites use KDFs for their password hashing needs. This is more than likely due to a lack of knowledge on cryptographic hashing functions or negligence on the developer’s end. Furthermore, these results complement Appendix A showing the prevalent use of MD5 and SHA-1.

3.6 Current Security Conclusions

In summary of this literature's research objectives, the following conclusions can be established. According to the user password complexity studies, the password selection schemes a majority of users employs result in weak passwords. With an average password length of eight characters consisting of only lower alphanumeric characters, the risk factor with these associated accounts can become very severe due to the nature of fewer combinations available, as opposed to the other character sets. To counteract an attacker from the user's perspective, a combination of the characters from all the character sets, with a significantly longer password, can mitigate the risk of vulnerability. The mentioned methods such a PsychoPass and lengthy password generation can accomplish this goal.

Based on the research analysis, the cryptographic hashing algorithms MD5 and SHA-1 are not acceptable algorithms for current systems to securely store passwords. The two algorithms weaknesses to collisions are not a prominent factor in this reasoning. Simply, the algorithms throughput for hash value generation, memory-conservation, the vast amount of data available to assist in cracking the hashes, and the short output length of their respective hash values. When these aspects become apparent to cryptographic hashing algorithms, an attacker is able to compute the hashes of large quantities of passwords per second. This is especially true when using hardware like GPUs, where all eight-character lower alphanumeric passwords could be cracked in mere hours. With respect to weak passwords, the severe risk increases dramatically when these passwords are hashed with weak cryptographic hashing algorithms. The overall extent of protection varies from which hash function is used based on a time-memory trade-off.

In regards to the developer's end, one can effectively store and secure passwords as a secondary defense mechanism to improve the overall security of a system by applying salt or pepper values to the password hashes, modifying the implementation of the cryptographic hashing algorithms to become iterative achieving a higher duration of calculation time for hash values, enforcing strong password requirements, and through implementing a KDF as opposed to the standard algorithms. Consequently, the analyzed attack methods on password hashes such as brute force, dictionary, advanced dictionary, lookup tables, reverse lookup tables, and finally rainbow table attacks will become ineffective or completely futile ensuring the highest degree of protection in the event of a comprise. In the next chapter, the latest cryptographic hashing algorithm will be discussed, as well as the current advancements and the future of authentication.

Chapter IV Future Authentication and Recent Developments

Year after year, requirements for passwords have been an ever-changing part of information security as hardware and software evolve over time. As Gordon Moore theorized in his famous law, computers become faster over time, human brains do not. The time it takes to break a given password hash is directly correlated to the current speed of computational processing. In other words, the strength of a cryptographic hash function can be measured by the number of hashes, which can be generated for plaintexts in a given period. As time goes on, the functions become weaker due to computing devices becoming faster and cheaper allowing powerful computing resources to be much more accessible. Thus, it is natural these types of algorithms become inadequate throughout their lifespan. In order to compensate, a new generation of more complex cryptographic hashing functions must be designed and produced when a predecessor becomes insufficient for password security needs. The following sections will present the most recent cryptographic hashing algorithm, current authentication advancements, and the future of authentication, respectively.

4.1 The Latest Cryptographic Hashing Algorithm

On November 2nd, 2007, a public competition was announced by NIST to develop a new cryptographic hashing algorithm to be called SHA-3, for standardization. The purpose of the competition was a response to the advancements of cryptanalysis of hash algorithms. These advancements were namely the attacks in 2004-2005, where several cryptographic hashing algorithms were attacked; in addition to serious attacks, being publicized against NIST approved SHA-1. Furthermore, in the event the SHA-2 algorithms were compromised, SHA-3 would be readily available to take its place. The competition ended on October 2nd, 2012, where the announced winner out of 64 entries was KECCAK to be standardized as the new SHA-3. During 2014, NIST published the Federal Information Processing Standard 202 (FIPS 202), “SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions” [51] [54], where on August 5th, 2015, FIPS 202 was approved indicating SHA-3 would be a hashing standard [53].

4.1.1 SHA-3 and KECCKA

The SHA-3 family shares identical functions of the SHA-2 family, namely: SHA3-224, SHA3-256, SHA3-384, and SHA3-512, where each function is based off an instance of the KECCAK algorithm [52]. With SHA-2, SHA-3 shares the same output scheme where the three-digit number in the function's name represents its output length in bits. "In specific, KECCAK is a family of sponge functions. The sponge function is a generalization of the concept of cryptographic hash functions with infinite output and can perform quasi all symmetric cryptographic functions, from hashing to pseudo-random number generation to authenticated encryption" [55].

At a high level, a sponge function shares the same functionality of a physical sponge, but with data. KECCAK being the sponge absorbs data into itself then; the data is eventually "squeezed" out, where the resulting output from the sponge is the hash value. The sponge implementation is unique in its design, as the traditional cryptographic hashing algorithms manipulate data through a more chronological approach.

Furthermore, the designers of KECCAK mention three primary strengths of the algorithm being: flexibility, design, security, and implementation. These benefits were designed with the intent to withstand various attack methods against the algorithm. Details of these strengths can be seen at [55]. Most notably, the implementation strength is said to excel in hardware performance where in comparison to SHA-2 it can outperform the function by an order of magnitude. On Modern PC's the algorithm is faster compared to the SHA-2 family and performs outstandingly when parallelism can be exploited.

As concluded in the previous chapter, cryptographic hashing algorithms are typically optimized and fast unlike KDFs. SHA-3 may be very suitable for collision resistance in terms of a primary strength, although, in terms of password security, the developers implementing SHA-3 for a system cannot rely on their users to formulate secure passwords. Additional measures as discussed in section 3.4 would need to be applied as supplementary defense measures to protect user password hashes at the highest degree.

4.2 Current Authentication Advancements

In accordance with the *Oxford English Dictionary*, the definition of authentication is "prove or show (something) to be true, genuine, or valid, or more simply, have one's identity verified." Based on this definition, it can be said passwords are a form of one-factor authentication technology where all, which is needed to validate a person, is the password itself. Thus, two-factor authentication (2FA), or two-step authentication, as the name suggests, is a technology where the identification of a user is based on the combination of two components. The components could range from what the user may know, a possession of the user, or anything, which is inseparable from the user. A basic example would be purchasing goods from a store; only the correct combination of a bank card and the associated personal identification number (PIN) will allow a purchase to proceed at checkout. 2FA is not new or recent; it is over a couple decades in age as the technology was patented in 1984 [56]. The age of the technology is not relevant; the significance of the technology lies in the various possibilities for unique 2FA applications, ultimately adding an extra security layer to protect users.

4.2.1 YubiKey

YubiKey, manufactured by Yubico is a hardware authentication device primarily used for the application of 2FA. The big name customers of the YubiKey range from Google (users and employees), Facebook (employees), the Department of Defense, GitHub, Duke University, and many other large software firms [58]. Specifically, a YubiKey consists of circuitry encased by an injection of a plastic molding where the exposed elements on the tokens consist of military grade hardened gold. The purpose of these physical attributes is to protect the key from internal damage or attempts to duplicate or record the token. A YubiKey is only powered through a universal serial bus (USB) port as the device itself is in the absence of batteries [57] [58]. The fourth generation YubiKey launched on November 16th, 2015 is shown in Figure 4-1.

Figure 4-1 A YubiKey [57]

To securely allow users to log into their accounts, the YubiKey emits a one-time password (OTP) which is verified by the service. No drivers or special software is required for the YubiKey. Once the YubiKey has been configured with the desired account, specific information about the key is stored which is later used to validate the OTP upon subsequent logins. Web sites, which do not support OTP, can still integrate the YubiKey as 2FA because the key also allows for the storage of static passwords. According to Yubico [58], the information that makes up a YubiKey OTP consists of:

1) The private identity of the YubiKey.
2) Counter fields tracking how often the YubiKey has been used.
3) A Timer field is tracking the time between generating each OTP.
4) A Random number to add additional security to the encryption.
5) A closing CRC16 checksum of all the fields.

Since Google allows 2FA with the YubiKey for their users, an individual would log in to their Google account by first entering the associated username and password combination, once validated, the service will request proof of the presence of the associated YubiKey for the account. The individual would then insert their YubiKey to a USB port on their computer. Authentication is finalized when the user presses the button on the YubiKey, emitting the generated OTP. After the OTP is validated based on its properties mentioned above, the user will be logged into their account successfully and securely.

The primary security benefit of the YubiKey is a user must have the key on them in order to log into a system. In the event a user's password is compromised, the attacker would not be able to access the account unless they possessed the associated YubiKey. Granted this benefit, the user's account would remain secure while giving the user the ability to reset their password afterward without any damage being done. Furthermore, any potential attack method on the password's hash value becomes negligible.

As YubiKey is a recent technology, but gaining popularity, only a handful of Internet Services supports the use of YubiKey such as Google Accounts, GitHub, Dropbox, LastPass, WordPress, and Code Signing. Regardless of operating system (OS), the YubiKey can be used to strengthen authentication on Windows, Mac OS X, and Linux. Other uses would be adding an extra layer of security to Password Managers, and Disk Encryption [58].

4.2.2 Nymi

Developed by a Toronto-based company known as Bionym, the Nymi is a lightweight wristband, which identifies and confirms a user through electrocardiogram (ECG) sensors that monitor heartbeat. The device was released early September 2014. Figure 4-2 illustrates a Nymi wristband.

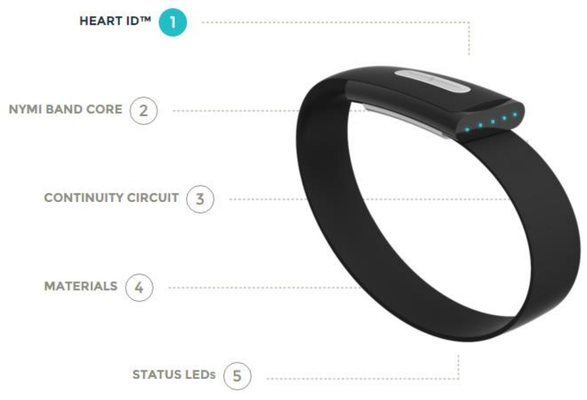


Figure 4-2 A Nymi wristband [66]

The information gathered by the wristband is transmitted to devices ranging from iPads to cars. To register an identity, all a user needs to do is simply touch a sensor embedded in the top of the band for 2 minutes. During this time, the Nymi captures an ECG signature, where once stored will only recognize the unique biometric template. Once the signature is captured, a user can access a device by touching the top of the wristband for several seconds. As another sensor makes contact with the user's wrist and retrieves its required data, it will communicate via Bluetooth to an app running on the device.

The Nymi also supports gestures, including waves and flex of the wrist, to be used when unlocking specific doors of a car for example. After Nymi collaborated with Toronto-Dominion (TD) Bank Group and MasterCard, the wristband can now be used to authenticate payments as a wearable credit card. Once the wristband is removed, devices all devices linked to it will automatically lock [64]. Essentially, the band acts an extension of the user providing continuous authentication of devices and services, which are supported via Bluetooth [65].

4.3 Project Abacus the Future of Authentication

Project Abacus is one of the most recent undertakings by Google's advanced technology and projects group (ATAP), announced last year in May at Google I/O [62]. The project's focus is to redefine the current approach to user authentication, such as passwords and 2FA primarily for smartphones. In other words, it's Google's attempt to depreciate the password with an entirely new form of user authentication.

Project Abacus' aim simply said, is to make a user entity the password itself by replacing the password with human interaction [61] [62]. The current research is a proposition to genuine multi-factor authentication, incorporating a variety of factors combined such as location patterns, how a person talks, how a person walks, voice and speech patterns, an individual's face, amongst other factors to identify a user. All this forms a unique usage pattern (trust score) for every user who becomes incredibly hard to fake for unwanted parties [61]. Unlike a password, this form of identification authenticates based on who the person is, not what they type.

An obvious observation would be the dramatic difference in reliability as opposed to fingerprints, passwords, four-digit PINs, etc. The team mentions if for any reason the legitimate user cannot be identified based on their trust score, the alternate method to authenticate would fall back on requesting the password associated with the application. Most significantly, this proposed technology requires nothing more than a software update; no specific hardware is required, everything needed for operation currently exists in every modern smartphone [62].

4.4 Conclusions on Future and Current Authentication

This chapter discussed the most recent cryptographic hashing algorithm used as a standard today as well as current authentication advancements and the future of authentication for smartphones/mobile devices. Without question, mobile devices today, are an ever-growing part of everyone's life, and as the focus on different authentication means is clearly directed to these devices. It is only logical as individuals become more reliant on their devices. Thus, security becomes an important growing concern.

On the other hand, aside from the YubiKey and the Nymi, authentication of user accounts regarding various websites has not seen significant innovations. Due to the nature of the overhead and challenge of identification from a client-server perspective, aside from the password, developers would need to integrate 2FA such as the YubiKey to provide their users an additional layer of security. Otherwise, he or she would need to create their own authentication standard, which is highly ill-advised, as it is recommended to integrate methods approved and standardized. Until a new revolutionary standard is designed to completely deprecate the password, it will remain as the primary standard for authentication.

Chapter V Attack Method Implementations

This chapter will present the developed implementations of three attack methods mentioned in Chapter 3 to further illustrate the weakness of MD5 and SHA-1. The first section of this chapter will discuss the purpose of the implementations, the second section discusses the technology used to develop the attack methods, the third, fourth, and fifth section discusses the attacks, examines the steps taken for development, and conveys the accomplishments of the approaches with respect to the thesis. The last section is a conclusion on the results.

5.1 Purpose

As the primary objective of this literature is to bring insight on why MD5 and SHA-1 are no longer suitable for securely storing password hash values, three attack methods, namely a dictionary attack, rainbow table attack, and brute force attack have been implemented to acquire significant results regarding time complexity and storage with each algorithm. The acquired data will be compared to the algorithms SHA-256 and SHA-512, which are suitable for securing passwords. In real environments, these attacks are the most commonly employed by individuals attempting to crack hash values, thus, they are suitable for implementation and demonstration. The steps taken for implementation of each attack will be explained in detail for the added benefit of comprehension for the reader.

5.2 Technology

The technology/software utilized for development of the attack methods were Hypertext Preprocessor (PHP), Hypertext Markup Language (HTML), Cascading Style Sheets (CSS), JavaScript, Java, XAMPP (Operating system, Apache, MySQL, PHP, and PERL), Large Text File Viewer (LTF), and RainbowCrack 1.6.1 (various packaged programs). Using these various technologies/software, a web application offering an interface to dictionary attack was implemented. The application allows a user to input either MD5 or SHA-1 hash values. Once the values are verified, a third party Java program was developed to perform a binary search across multiple files searching for the values. Post-search, the web application displays the results to the user.

Regarding the rainbow table attack method, the software known as RainbowCrack allows generation and searching of rainbow tables using one of various programs included in the software package. Specialized programs in the package utilize GPU acceleration as a means of searching. The primary benefit of this software package is the platform friendliness and the maximum utilization of the specified resource(s); different programs were included for GPU's such as AMD and NVIDIA, and if a GPU was not present, another program was present to maximize utilization of the machine's central processing unit (CPU) for searching through the generated tables.

Lastly, the brute force attack method was implemented with a generic brute force attack running on a single core of execution, and a sophisticated machine dependent brute force scaling attack, involving multithreading to take advantage of all available processors in a given system. These implementations, were developed using Java. Both brute force attacks acquired results which are used for comparison purposes to the associated cryptographic hashing algorithms.

5.3 Dictionary Attack Implementation

This attack method is the first course of action an attacker would undertake on a set of given hash values, in order to firstly eliminate the weakest passwords in the set. To implement such an attack, four primary steps are required.

The first step to creating a dictionary attack is gathering a set of candidate plain text passwords with the highest potential of success. As the purpose of this attack is to exploit users who employ weak password schemes such as including words, common phrases, using similar or the same passwords across various accounts, or appending well known strings to the passwords such as "123", any plain text with some sort of corresponding meaning is acceptable. Generally, any words, phrases, common combination of words, geographic locations, places, brands, and names should be included in the attack files. The larger the set of files including these candidate plaintexts, the higher probability a given hash value will be cracked when searching through them. Specifically, the entries in the researcher's attack files include, but are not limited to:

- Leaked passwords from Rockyou.com (2009), MySpace.com (2006), Hotmail.com, Faithwriters.com, Elitehacker.com, Facebook.com (2010), and Singles.org (2010).
- Leaked Facebook first and last names (over 100 million).
- World cities, countries, states, and other geographic locations.

- The English, Italian, and Polish dictionary (Over 3 million total words).
- The top 100 password lists of the last decade.
- Word lists used by popular password cracking software such as John the Ripper and Cain and Abel.

In summation, 18 gigabytes (GB) of data was assembled after five hours of searching and downloading. This resulted in over 1.3 billion entries in total. It is worth mentioning this amount of data for a dictionary attack is quite small-scale as large-scale dictionary attacks can easily reach up to 200 GB worth of plaintexts (11 billion entries). Once all candidate entries are gathered, it is best practice to merge all the files together for simplicity. Merging is accomplished by simply using the command "*for %f in (*.txt) do type "%f" >> output & echo: >> outpu*t" in the respective directory of all the files through command prompt. Through this, all entries will be on separate lines. With this amount of data, the merging process required three hours of processing.

Post completion of merging, the second step is to generate and store the entries' respective hash values with the cryptographic hashing algorithm(s) of choice. This step's reasoning is to create a non-sorted lookup table of all the entries. For the purpose of this literature's focus, generation and storage of the corresponding MD5 and SHA-1 hash values for each entry was undertaken. Using the programming language Java, a customized program read in all 1.3 billion entries line by line, then, generated and stored the mapped hash values for each algorithm in two separate text files in separate folders as (Hash Value, Plain text) pairs.

The entire process of generating the values for each algorithm completed after six hours of computations, the MD5 file was 62 GB while the SHA-1 file was 73 GB. With text files of this magnitude, specialized software is required to view each file's contents. A program known as Large Text File viewer (LTF) is designed to meet these need and was applied when required.

To further increase the probability of success in a dictionary attack, an attacker at this point can regenerate, then rehash the original plain text file with all uppercase letters, lowercase letters, a mix of both cases, "lees speak" representations of the plaintexts (Advanced Dictionary Attack), and make any appending or prepending to the entries. The significance is to cover as many different variations of the original plain text file. Regarding the storage impact on the inclusion of the mentioned variations, the summation of data can easily reach five times more hard disk space and computational time. For instance, if the mentioned variations for both MD5 and SHA-1 were taken, the hard drive space needs would rise to roughly 816 GB from 136 GB. Although, with the cheap cost of hard drives today, this is negligible in comparison to the additional time required for generation. The original file must first be regenerated for the variation, then the consequent file would need to be rehashed for both MD5 and SHA-1. The time for this process could reach up to thirty hours of processing. It is worth noting the lengthy time duration arises from the CPU being limited in utilization due to the speed of writing and reading from the hard drive.

Once the lookup tables are completed, the next step is to sort all the entries. This step is necessary to achieve optimal conditions for performance when searching through files of this size degree. A sequential search is not ideal as one hash value could take an extended duration to find with this much data. Using a Java implementation, another customized program read in all entries from the massive lookup tables in a binary search tree (BST), then wrote the entries to a file using a recursive in order traversal of the BST. One primary benefit is all duplicates within each file were removed.

With a limitation of 8 GB of DDR3 random access memory (RAM), the researcher chose to break up the massive lookup tables into 130 unique text files by loading 10 million entries at a time in the BST. Since there is a limit to the amount of heap-space the Java Virtual Machine assigns to a Java program, a bypass is needed to load exactly ten million entries at a time. To accomplish this, the program can be executed through the command line. The amount of memory issued by the argument "–Xmx7g" (Max heap space = 7 GB) satisfied this requirement. Again, this process occurred twice, once for the MD5 and the SHA-1 text file, totaling five and a half hours of processing time. Afterward, with the set of sorted text files, an additional program performed an external merge and sort on the set of files into one large sorted file for both MD5 and SHA-1. The entireties of all duplicates in the files were removed, leaving all unique hash value candidates. Finally, the large files were once again broken down into pieces of 10 million. The first and the last hash value of each file were used to index the files for searching.

The fifth and last step is implementing a searching algorithm for the sorted lookup tables. The method of choice was employing a binary search directly on the text files. Applying an implicit search after loading the values would be inefficient because as all values are loaded into memory, a passive sequential search is undertaken. To overcome this, a binary search was implemented on the text files externally, using Java’s random access file class. This object allowed the seeking to specific byte locations. When searching through the records in all scenarios, namely best, average, and worst, a time complexity of O (log (n)) was achieved. At most, a single file would only require 29 iterations.

The web application portion of the dictionary attack was used to receive input hash values of MD5 or SHA-1 from a user, and then relay the results in a table to determine if a given hash was present in the dictionary attack files. PHP, CSS, HTML, and JavaScript were used to design the graphical user interface and validate input. Once the hash values were verified, the PHP side of the application would pass the values as arguments to a jar file (The Dictionary Attack), which would perform the search then return the output results back to PHP to be displayed using HTML. The next section discusses the results of the attack. It is worth mentioning the reasoning for storing the hash values into text files was for the basis of time and space in comparison. In real environments, a database would be utilized, as search times would be dramatically reduced.

5.3.1 Dictionary Attack Results and Comparison

The relevant results from the implementation of a sequential file search and binary search on the associated dictionary attack files, as discussed in the previous section, are illustrated in the tables below. The sequential search was conducted on two non-sorted text files containing all 1.3 billion entries for MD5 and SHA-1 with their associated plaintexts. The binary search was applied to 130 sorted text files consisting of 10 million entries each with their mapped hash value/plain text pairs. The results of SHA-256 and SHA-512 are estimated based on the factor of additional work required with respect to SHA-1. Since a SHA-1 hash value consists of 40, while a SHA-512 hash value consists of 128, it can be said a SHA-512 hash value requires 3.2 times more disk space and time for searching in the scenario of a dictionary attack with both a sequential and binary search.

Hashing Algorithm	Number of Hashes to Search	File Size (Approx.)	Number of Iterations	Time (Approx. Worst case)	Time Complexity (Worst Case)
MD5	10	63 GB	1.3 billion	8.4 min	O(n)
SHA-1	10	73 GB	1.3 billion	15.6 min	O(n)
SHA-256	10	116.5 GB	1.3 billion	25 min	O(n)
SHA-512	10	233 GB	1.3 billion	50 min	O(n)

Table 5-1 Sequential search results on a non-sorted text file

Hashing Algorithm	Number of Hashes to Search	Size of each File (Approx.)	Max. Number of Iterations per File	Time (Avg. Worst case scenario)	Time Complexity (Worst Case)
MD5	1000	460 MB	29	40-55 seconds	O(log(n))
SHA-1	1000	537 MB	29	50-70 seconds	O(log(n))
SHA-256	1000	860 MB	29	80 – 112 seconds	O(log(n))
SHA-512	1000	1.7 GB	29	160 – 224 seconds	O(log(n))

Table 5-2 Binary search results across 130-sorted files

Hashing Algorithm	Required Disk Space Per Hash Value	Total Disk Space Required (+1 byte for comma separator)
MD5	32 Bytes	(32 * Number of Hashes) + (Σ plain text lengths + 1)
SHA-1	40 Bytes	(40 * Number of Hashes) + (Σ plain text lengths + 1)
SHA-256	64 Bytes	(64 * Number of Hashes) + (Σ plain text lengths + 1)
SHA-512	128 Bytes	(128 * Number of Hashes) + (Σ plain text lengths + 1)

Table 5-3 Required disk space for each algorithm

In terms of time complexity based on these results, the degree of performance differentiation with a binary search of files of this size magnitude in comparison to sequential searching is clearly illustrated. The event an attacker acquires a large set of compromised hash values, and does not have a database as a resource, a binary search is a necessity to locate the largest number of hash values in the shortest amount of time. With respect to MD5 and SHA-1, SHA-256 and SHA-512 requires a negligible amount of additional time to search through the files with a small set of hash values. Although, in an attack scenario with 1 million compromised hash values, utilizing the indexed binary search implementation would require approximately eleven hours with MD5 in comparison to the sixty-two hours required by SHA-512.

Space complexity, on the other hand, has a significant impact on SHA-512 as 3.2 times more disk space is needed to store the same number of hash values of SHA-1 and 4x more than MD5. If an attacker were to include the five mentioned variations of the original plain text file, namely, uppercase letters, lowercase letters, a mix of both cases, “leet speak” representations of the plaintexts, and appending or prepending to the entries, the amount of disk space required for MD5 values would be 315 GB in comparison to 1.2 terabytes (TB) required by SHA-512. Again, this attack is a small scale, an attack scaled by a factor of ten would require over 10 TB of memory for SHA-512.

Overall, the time of searching and the storing of hash values with a dictionary attack using this implementation differ greatly between the algorithms with large sets of data. An attacker will always be able to store 4 times more MD5 and 3.2 times more SHA-1 data in comparison to SHA-512. With today’s technology, it is evident SHA-512 is a much stronger alternative than MD5 and SHA-1.

5.4 Rainbow Table Attack Implementation

Once an attacker filters out the unmasked hash values from the set, the second course of action is a rainbow table attack. This attack’s most notable strength is achieving greater than a 99% chance of finding a given hash value. Although, this assumes the original plain text consists of characters in the character set used to generate the rainbow tables. To implement this attack project RainbowCrack’s downloadable software package was employed. The three primary programs required from the package was the rainbow table generator, rainbow table sorter, and the graphical user interface (GUI) for input and searching through the tables. With this approach, only three simple steps are required. The Windows operating system will be referenced for this implementation.

Once the software package is downloaded, a user needs to navigate to the unzipped directory of the folder through the command prompt, and then type a specific command to start generation of a single rainbow table. The commands required to generate MD5 and SHA-1 rainbow tables with the character set of lower alphanumeric, password lengths 1-8, are “*rtgen md5 loweralphanumeric 1 8 0 10000 33554432 0*” and “*rtgen sha1 loweralphanumeric 1 8 0 10000 33554432 0*” respectively. Explanations of the commands are as follows.

- Parameter 1 (rtgen): Specifies a rainbow table is to be generated.
- Parameter 2 (md5/sha1): Specifies the cryptographic hashing algorithm for hash value generation.

- Parameter 3 (loweralphanumeric): Specifies the charset to be used during generation. Any set in Appendix E can be applied.
- Parameter 4/5 (1 8): Specifies the lengths of the password's plaintexts, which will be in the table.
- Parameter 6 (0): Specifies the rainbow table number.
- Parameter 7 (10000): Specifies the length of each chain in the table.
- Parameter 8 (33554432): Specifies the number of chains in the table.
- Parameter 9 (0): Specifies a file descriptor. This is the only part of the filename, which can be changed.

After the command is executed, generation of the table will begin. With these particular commands, the generation time of a single table for MD5 and SHA1 was three and four and a half hours, respectively, using an Intel(R) Core(TM) i7-4470K CPU @ 3.50GHz at 100% utilization.

Since the keyspace chosen was passwords of length 1-8 lower alphanumeric, the summation of possibilities in this space is 2,901,713,047,668 (36^1 + 36^2 + 36^3 + 36^4 + 36^5 + 36^6 + 36^7 + 36^8). With the rainbow table specifications in my implementation, a single table would cover 335,544,320,000 possibilities, meaning to cover the entire keyspace; nine tables would need to be generated in theory. As this is impractical due to duplicates across the tables, the actual chance of success in this keyspace would be slightly less than 40% with only nine tables. Therefore, achieving a 100%, chance of success is indeterminable because a great degree of difficulty is associated to cover the remaining possibilities not included in the table past a 90% success chance. The highest success chance achieved in the researchers' scenario was 93%. The set of 25 MD5 and SHA-1 tables involved 7.8 days of processing time across ten Intel(R) Core(TM) i7-4470K CPU @ 3.50GHz at 100% utilization. Once the processing was completed, a total of 24.5 GB was required to store the 50 tables equaling 0.877 petabytes abstractly stored. The following calculation estimates the amount of data abstractly stored.

- (SHA-1) 335,544,320,000 (possibilities per table)/1,300,000,000 (comparison value based on the number of entries in the dictionary attack) = 258.11*73 GB (storage requirement of 1.3 billion entries based on the dictionary attack) = 18842.03 GB = 18.8 TB/ table * 25 (total number of tables) = 471 TB

- (MD5) 335,544,320,000 (possibilities per table)//1,300,000,000 (comparison value based on the number of entries in the dictionary attack) = 258.11*63 GB (storage requirement of 1.3 billion entries based on the dictionary attack) = 16260.93 GB = 16.2 TB/ table * 25 = 406 TB
- 471 TB + 406 TB = 877 TB
- 877 TB = 0.877 Petabytes

Furthermore, for research purposes, an Amazon Web Services (AWS) account was created to test the time complexity and cost of generating the same number of SHA-1 tables on a c4.8x large instance with the following specifications: a cluster of thirty-six vCPUs being Intel Xenon high-performance processors (2.9GHz), 10 GIGABIT Network Connection, and 60 GB RAM. In two minutes and nineteen seconds, the instance generated 1,179,648 of 33,554,432 entries, while the machines used by the researcher with SHA-1 generated slightly over 200,000 entries in the same amount of time. The time to generate a single SHA-1 table in this instance is 62 minutes costing $3.35 US ($1.675/hour). Overall, the 25 SHA-1 rainbow tables generated could have been generated for 42$ US in slightly over a day. If the attacker was well funded, they could easily use AWS to generate rainbow tables with a 90% success chance for longer length passwords and various character sets.

After all the tables have completed generation, the next step is sorting them using the rtsort <filename>.rt command in the RainbowCrack directory where the generated tables are stored. Sorting a single table takes no more than a few moments. Lastly, the final step is searching the tables for given hash values, which can be inputted through the GUI of the desired program based on system specifications. A separate GUI is present for AMD and NIVIDA GPUs as well as a CPU. By default, every program will utilize the specified resource to its full potential. With GPU acceleration, the 25 tables for either MD5 or SHA1 can be searched through entirely in seconds. The next section discusses the results of the attack.

5.4.1 Rainbow Table Attack Results and Comparison

The following results were derived from 1000 hash values of MD5 and SHA-1 obtained through randomly generating plain text with the lower alphanumeric character set. Both sets of hashes mapped to the same plaintexts to effectively establish a basis for comparison between the set of rainbow tables. The GPU of choice for searching through the tables was an NVIDIA GeForce

GTX 660, utilized by RainbowCrack's specialized NVIDIA GPU program utilizing the graphics card to its maximum capabilities.

Hashing Algorithm	Time Taken to Generate One Table	Time Taken to Generate 25 Tables	Hard Disk Space to Store Tables	Success Chance
MD5	3 hours	3.1 days	12.5 GB	93.7%
SHA-1	4.5 hours	4.6 days	12.5 GB	92.2%
SHA-256	10 hours	10.4 days	12.5 GB	92%-93%
SHA-512	20 hours	20.8 days	12.5 GB	92%-93%

Table 5-4 Rainbow table generation statistics

Hashing Algorithm	Time to Search 1000 Hash Values	Time to Search 10,000 Hash Values	Time to Search 100,000 Hash Values	Time to Search 1,000,000 Hash Values	Time to Search 10,000,000 Hash Values
MD5	10.60 min	1.7 hours	17 hours	7.08 days	70.8 days
SHA-1	19.05 min	3.1 hours	1.29 days	12.9 days	129.0 days
SHA-256	30.48 min	5.08 hours	2.11 days	21.1 days	211.0 days
SHA-512	60.96 min	10.16 hours	4.23 days	42.3 days	423.0 days

Table 5-5 Rainbow table search times

As rainbow tables only need to store two strings per chain, the amount of data required for each table is roughly the same. Although, with the same amount of data being searched, each algorithm takes a significantly longer duration of time due to the extra memory required for comparisons. Since a SHA-256 hash value contains 1.6 times more characters than a SHA-1 hash value, there is a direct correlation between the processing needed to be performed while searching. With this relation, a SHA-512 hash value requires 3.2 times more processing than a SHA-1 hash value.

Through analyzing these results, the difference between time complexities is quite variable between MD5 and SHA-1, in comparison to SHA-512. The time-memory tradeoff is accurately shown in this scenario; more tables increase the chance of probability at the tradeoff of increased time to search through the set of tables. With a keyspace of 2,901,713,047,668 in the given password length range and the specified character set, a single table covers 335,544,320,000 possibilities. Thus, 25 rainbow table's results in 8,388,608,000,000 possibilities covered. This is almost three times more combinations than that of the original keyspace due to duplicates. The additional data are a necessity in order to effectively achieve over a 90% success ratio.

As an outcome, if a well-funded attacker had five times the amount of resources in this scenario, the time complexity of generation and searching could be cut down by five times, respectively. One million MD5 and SHA-1 hash values could be searched in 1.4 days and 2.5 days respectively while 1 million SHA-512 hash values would require 8.46 days. In scenarios with larger character sets and password lengths, the time complexity of each algorithm would be exponential. With this, data on the performance results of the algorithms it is quite evident MD5 and SHA-1 require dramatically less time to crack their hash values in comparison to SHA-512 with the technological resources available today.

5.5 Brute Force Attack Implementation

The last resort an attacker would employ against a set of masked hash values would be a brute force attack. This attack is a last resort because of the time complexity required to reveal a single hash value due to the extreme magnitude of basic operations required while iterating. In order to brute force through a six character, lower alphanumeric password, in the worst-case scenario 2,176,782,336 iterations are required, in addition to the number of iterations needed to generate a hash value for a given plain text. At a high level, a hash value is generated on a character-by-character basis, meaning an MD5 hash value would require 32 iterations, SHA-1 forty iterations, and SHA-512 128 iterations for any password. Thus, 87,071,293,440 iterations in total would be required to brute force a SHA-1 hash value with the specified password criteria. Despite this, unlike the other attack methods the brute force approach guarantees a successful result as all possibilities are searched. The resources required for iterating, in addition to time, are the only significant factors.

Specifically, my brute force attack involved two separate implementations in Java for comparison purposes. The user input for both programs were the hash value(s) represented in a string array and the character set represented by a character array. The keyspace based on password length to be searched is hard coded as six characters for practicality. Implementation one involved a simplistic/generic brute force attack running on a single thread of execution where the time complexity in the worst-case scenario was recorded. The second implementation was a more sophisticated approach. The program utilized multithreading to divide the keyspace of a character set into pieces based on the number of processors (physical/logical) available on a given system. Once divided, a separate thread of execution is assigned one piece of the division as its processing work. Based on my implementation the program is scalable to any system in order to achieve the fastest results possible. The following section covers the results of the attacks.

5.5.1 Brute Force Attack Implementation Results

Following is the results with respect to time complexity in the worst-case scenario on two brute force attack implementations shown in Table 5.6 and 5.7. The time required to iterate through the entire keyspace of length six passwords consisting of lower alphanumeric characters is shown. The technology utilized to its maximum potential in this attack was a hyper-threaded Intel(R) Core(TM) i7-4470K CPU @ 3.50GHz for both implementations. With the nature of brute force attacks typically being a sequential approach, no hash values were used to determine time complexity. Results would be highly dependent on the location of the password in the set of possibilities. Using a set of hash values as input for this attack method, the total number of iterations illustrated below would need to be multiplied by the number of hash values in the set. The statistics for SHA-256 and SHA-512 are not estimated based on a work factor as with the previous methods. They were explicitly determined through the program.

Hashing Algorithm	Time Per Letter Set (Approx.)	Total Time (Worst Case Approx.)	Time Complexity (Worst Case)	Number of Iterations
MD5	77 seconds	46.2 min.	O (n^6)	2,176,782,336 * 32
SHA-1	92 seconds	55.2 min.	O (n^6)	2,176,782,336 * 40
SHA-256	149 seconds	1 hour 30 min.	O (n^6)	2,176,782,336 * 64
SHA-512	311 seconds	3 hours 7 min.	O (n^6)	2,176,782,336 * 128

Table 5-6 Single thread brute force implementation results

Hashing Algorithm	Time Per Letter Set Per Thread (Approx.)	Total Time (Worst Case Approx.)	Time Complexity (Worst Case)	Number of Iterations
MD5	160 seconds	12 min.	O (n^6)	2,176,782,336 * 32
SHA-1	190 seconds	14 min. 15 sec.	O (n^6)	2,176,782,336 * 40
SHA-256	279 seconds	21 min.	O (n^6)	2,176,782,336 * 64

SHA-512	550 seconds	41 min. 15 sec.	O (n^6)	2,176,782,336 * 128

Table 5-7 Multi-threaded brute force implementation results (8 threads)

The primary significance of this implementation is the comparison of time complexity factors between SHA-1 and SHA-512. With the generic single threaded program, SHA-512 required 3.5 times more time to complete its iterations while with the multithreaded program, SHA-512 only required 2.1 times more time to complete its iterations. Despite this, in comparison to the other two algorithms SHA-512 will require a substantial amount of additional processing time to complete the same keyspace for that of a MD5 or SHA-1 hash value. With respect to Moore's law, as CPU's performance continues to increase, hashing algorithms will continue to become weaker. Thus, it would be more beneficial to implement SHA-512 as the cryptographic hashing algorithm in a system due to the significantly increased time complexity required to obtain the plain text of hash value, in comparison to the other algorithms. With a GPU implementation, the desired results would be relative; a SHA-512 or SHA-256 hash value will always require more processing time than the other two algorithms.

5.6 Conclusions on Attack Method Implementations

In conclusion of the implemented attack methods, the cryptographic hashing algorithms MD5 and SHA-1 have shown to perform far less than optimal when faced with the discussed attacks. The algorithms are not suitable, not because of any cryptographic weakness, but rather due to work factors, performance, and space complexity. On obsolete computers from over a decade ago, the algorithms were optimal because of the current technology available. Processing power was not an abundant and cheap resource as it is today. To compensate, the SHA-512 algorithm should be employed in systems as time and space complexity is fairly relative to the processing power available today.

As all the attacks were performed on a small scale with limited resources, consequently an environment for cracking MD5 and SHA-1 hash values was created. This is under the assumption the hashes compromised abided by the research on user password complexity where 33% of user's passwords could be obtained with the developed environment. In contrast to an attacker with an advanced and abundant amount of resources, the weaknesses of MD5 and SHA-1 in the respective context could be exploited dramatically.

To reiterate, the goal of password cracking is to unmask the largest number of passwords in the shortest amount of time. The more time required to decode hash values allows a greater duration of time for system administrators to implement countermeasures such as notifying users to change their passwords or to lock out accounts. This statement demonstrates the significance of users creating strong passwords. Since the implemented hashing algorithm is unknown to a user, the safest course of action is to create a lengthy and complex password, which will take an exponentially longer period of time to determine.

Chapter VI

6.1 Conclusion

Through research and statistical analysis from experiementation, this literary work has provided significant reasoning and insight on the insecurity of the cryptographic hashing algorithms MD5 and SHA-1. Results have illustrated the respective algorithms utilized for password masking are no longer acceptable for security measures regarding the hashing, confidentiality, and protection of passwords in today's systems and applications.

The representative weakness of both algorithms regards digital signatures. Collisions have been found for MD5 and theoretically potential collisions can be obtained for SHA-1. With respect to password hashing, both algorithms satisfy the cryptographic hashing function requirements to be considered applicable in practice. In essence, MD5 and SHA-1 are insecure not because of cryptographic weakness; rather, the non-applicability of the functions arises from suboptimal password generation habits of users. Thus, there is a direct correlation between the strength of a cryptographic hashing algorithm and the strength of a user's password. In other words, the output of a cryptographic hashing function is only as secure as its input plain text.

Implementation of attack methodologies against MD5 and SHA-1 has illustrated either algorithm's generated hash values require a dramatically smaller duration of time and space to determine as opposed to a more recent algorithm such as SHA-512. With this said, the following statement holds true: *Any given plain text hashed with the most recent cryptographic hashing function of the same output class will embody a higher degree of security to the identical plain text hashed with any prior cryptographic hashing algorithm.* Anticipating users will deliberately generate weak passwords; modern cryptographic hashing functions are required, as these mentioned passwords will be effectually protected to a greater degree of security. Although, improvements such as iterative hashing and salting can be augmented to the base MD5 and SHA-1 algorithms to better their applicably, it is rather futile as improving a modern base algorithm will inevitably offer an even greater extent of protection in all scenarios.

As time goes on, all cryptographic hashing algorithms become weaker due to lower costs of hardware and advancing technology. The result is a decrease in the work factor required to reveal passwords of a given length. With MD5 and SHA-1 being published over a decade ago and given user password generation habits as of today, the algorithms have well outlived their life cycles and applicability. On the other hand, if all users were to utilize a password of a very great length with all character types, MD5 and SHA-1 could be applicable for decades. Since this expectation of users is impractical due to human memory constraints, developers of username and password authentication schemes for a system should undertake such a matter with nothing less than a proactive approach. All in all, the degree of password security for a given system does not rely on a developer exclusively, but rather the degree is measured based on the summation of effort from both the developer and the system's users.

6.2 Future Work

Research with cryptographic hashing algorithms for future implementations correlated to password complexity studies can be further discussed. With the inevitability of computers becoming more sophisticated and cheaper over time, cryptographic hashing algorithms will become weaker. As a result, user passwords will need to be more complex to maintain a suitable standard for protection. Otherwise, if current user password habits remain, hash lengths of 256 bits or greater may need to be utilized to compensate. Alternatively, there could be a point in time where passwords will be considered obsolete and new methods to authenticate users will need to be employed in systems due to the impractically of users remembering multiple complex passwords of great lengths.

Glossary

Algorithm: An unambiguous process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer.

AND: The Boolean AND operation. In computer science, the operation will return true if all of the statements the operation is testing is true.

Bit: A binary digit: 0 or 1.

Bcrypt: A key derivation function for passwords.

Binary Search: A search algorithm used to locate the position of a target in a sorted list. As opposed to a classic search a binary search divides the list in half each iteration until the target is found.

Byte: A sequence of eight bits.

Character Set: A defined list of unique characters. A string generated using a character set will only consist of the defined characters in the set. See Appendix E for details.

Cryptography: The art of writing or solving codes.

Digest: The output of a cryptographic hash function. Also called the hash value.

Encryption: The transformation of data into another form typically known as the ciphertext, where the ciphertext cannot or should not be understood easily by anyone except authorized parties. The encrypted data is manipulated in such a way where decryption is possible with a secret key.

FIPS: Federal Information Processing Standard.

Freestart Collision: A collision where the attacker can choose the initialization vector (IV).

Function: Produces output based on the specifically defined input. A function is essentially a relation between its set of inputs and its associated corresponding output.

Hashcat: The world’s fastest CPU-based password recovery tool.

Hash Function: A function where the output is a set of bit strings in which the length of the output is fixed.

Hash Value: See digest.

Hashing: Converting a string of characters into a typically shorter value or key, which is usually fixed in size, which represents the original string. Used in many encryption algorithms.

HMAC: Keyed-Hash Message Authentication Code.

John the Ripper: A free password cracking software tool. Developed initially for the UNIX operating system, but now runs on 15 different platforms.

KDF: Key derivation function. A key derivation function, with respect to cryptography, creates one or more secret keys based on a secret key value whether it be a master key, a password, or a passphrase. Their primary use is for key stretching.

KECCAK: The family of all sponge functions with a KECCAK-f permutation as the underlying function and multi-rate padding as the padding rule.

Keyspace: The set of all possible combinations in a given range of password lengths.

Key Stretching: Techniques, which are used to convert possibly weak encryption key(s), a password, or passphrase into a form, which has a higher degree of security against brute force attacks. Through incrementing the duration of time taken to test each key this can be accomplished.

Message: A bit string of any length that is used as input for a cryptographic hashing function.

MD5: Message Digest 5.

NIST: National Institute of Standards and Technology.

OR: The Boolean OR operation. In computer science, the operation will return true if one or more statements the operation is testing is true.

OTP: One-time password

Output: Often serves as a condensed representation of the input.

Password: A string of characters that allows access to a computer, interface, or system.

PBKDF2: Password-Based Key Derivation Function 2 is a key derivation function that is part of RSA Laboratories’ Public-Key Cryptography Standards (PKCS) series.

Query: In computer science a query is known as a statement used to retrieve information from a database. A query can have many parameters specifying the exact data to be retrieved whether it be retrieving the entire record(s) or parts of record(s) which result from performing various calculations before the results are displayed.

Round: The sequence of steps, which is iterated in the computation of a hash value. Cryptographic hashing algorithms process many rounds before the final output.

SHA-1: Secure Hash Algorithm-1.

SHA-2: Secure Hash Algorithm-2.

SHA2-224: The SHA-2 hash function that produces 224-bit digests.

SHA2-256: The SHA-2 hash function that produces 256-bit digests.

SHA2-384: The SHA-2 hash function that produces 384-bit digests.

SHA2-512: The SHA-2 hash function that produces 512-bit digests.

SHA-3: Secure Hash Algorithm-3.

SHA3-224: The SHA-3 hash function that produces 224-bit digests.

SHA3-256: The SHA-3 hash function that produces 256-bit digests.

SHA3-384: The SHA-3 hash function that produces 384-bit digests.

SHA3-512: The SHA-3 hash function that produces 512-bit digests.

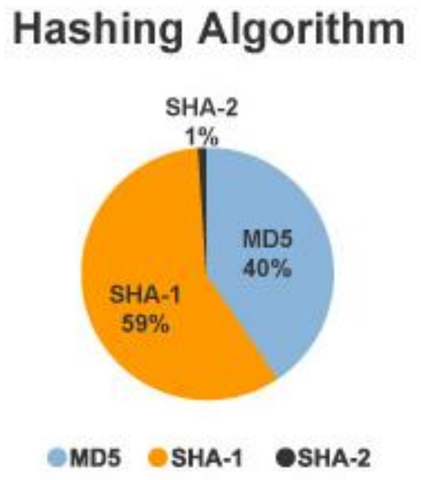
Slow Key Scheduler: A key derivation function property, which is iterative, with the intention to make brute force attacks against the algorithm less effective.

String: Any finite sequence of characters such as letters, numerals, symbols, and punctuation marks. The most significant property of string is its length, being the number of characters in it.

XOR: The Boolean Exclusive-OR operation. In computer science, the operation will return true if and only if one of the statements the operation is testing is true. In other words, if more than one statement is true the operation will return false.

Appendix A

MD5 AND SHA-1 USAGE STATISTICS



Based on Analysis of 2000 Global Organizations in 2013 [60]

This pie chart was created from research performed by Venafi, a privately held cybersecurity company who develops software to secure and protect cryptographic keys and digital certificates.

Due to the nature of information security, revealing the cryptographic hashing algorithm implementation is a major security breach. As there is minimal data regarding the statistics of the algorithms usage (disregarding the pie chart above), the popularity of the algorithms will be based on the websites discussed in section 3.5. Additionally, the most recent websites in the last 5 years having their hash algorithm revealed, specifically MD5 or SHA-1, are displayed below.

Website	Date	Cryptographic Hashing Algorithm(s)
Ashley Madison	July, 2015	Message Digest 5 and Bcrypt
EHarmony	June, 2012	Secure Hash Algorithm 1
LinkedIn	June, 2012	Message Digest 5

Revealed hashing algorithm of compromised websites in the last 5 years

Appendix B

THE 25 MOST POPULAR PASSWORDS OF 2014

Rank	Password	Change from 2013
1	123456	No Change
2	password	No Change
3	12345	Up 17
4	12345678	Down 1
5	Qwerty	Down 1
6	123456789	No Change
7	1234	Up 9
8	baseball	New
9	dragon	New
10	football	New
11	1234567	Down 4
12	monkey	Up 5
13	letmein	Up 1
14	abc123	Down 9
15	111111	Down 8
16	mustang	New
17	access	New
18	shadow	Unchanged
19	master	New
20	michael	New
21	superman	New
22	696969	New
23	123123	Down 12
24	batman	New
25	trustno1	Down 1

This password list was obtained from a password-management company known as Splash Data[6]. The company analyzed 3.3 million leaked password in 2014 forming a list of the most commonly used in the data set.

[6] Splash Data: www.splashdata.com

Appendix C

AN OVERVIEW OF THE MD5 ALGORITHM

MD5 works as follows on a given bit string M of arbitrary bit length [30]:

1. Padding. Pad the message: first append a ‘1’-bit, next append the least number of ‘0’-bits to make the resulting bit length equivalent to 448 modulo 512, and finally append the bit length of the original unpadded message M as a 64-bit little-endian integer. As a result, the total bit length of the padded message Mc is 512N for a positive integer N.

2. Partitioning. Partition the padded message Mc into N consecutive 512-bit blocks M0, M1, . . . , MN−1.

3. Processing. To hash a message consisting of N blocks, MD5 goes through N + 1 states IHVi , for 0 ≤ i ≤ N, called the intermediate hash values. Each intermediate hash value IHVi is a tuple of four 32-bit words (ai , bi , ci , di). For i = 0 it has a fixed public value called the initial value (IV): (a0, b0, c0, d0) = (6745230116, efcdab8916, 98badcfe16, 1032547616). For i = 1, 2, . . . , N intermediate hash value IHVi is computed using the MD5 compression function described in detail below: IHVi = MD5Compress(IHVi−1, Mi−1).

4. Output. The resulting hash value is the last intermediate hash value IHVN , expressed as the concatenation of the hexadecimal byte strings of the four words aN , bN , cN , dN , converted back from their little-endian representation. As an example, the IV would be expressed as 0123456789abcdeffedcba987654321016. Full details on the MD5 algorithm can be viewed at [30].

Appendix D

ADDITIONAL MD5 COLLISIONS

Message 1: [31]

Common MD5 Hash Value: 008ee33a9d58b51cfeb425b0959121c

4dc968ff0ee35c209572d4777b721587d36fa7b21bdc56b74a3dc0783e7b9518afbfa200a8284bf36e8e4b55b35f427593d849676da0d1555d8360fb5f07fea2

4dc968ff0ee35c209572d4777b721587d36fa7b21bdc56b74a3dc0783e7b9518afbfa202a8284bf36e8e4b55b35f427593d849676da0d1d55d8360fb5f07fea2

Message 2: [59]

Common MD5 Hash Value: cee9a457e790cf20d4bdaa6d69f01e41

0e306561559aa787d00bc6f70bbdfe3404cf03659e704f8534c00ffb659c4c8740cc942feb2da115a3f4155cbb8607497386656d7d1f34a42059d78f5a8dd1ef

0e306561559aa787d00bc6f70bbdfe3404cf03659e744f8534c00ffb659c4c8740cc942feb2da115a3f415dcbb8607497386656d7d1f34a42059d78f5a8dd1ef

Note: The examples above are a hexadecimal representation of the strings. In order to test it, these values must be written into binary files then the comparison can be accomplished.

Appendix E

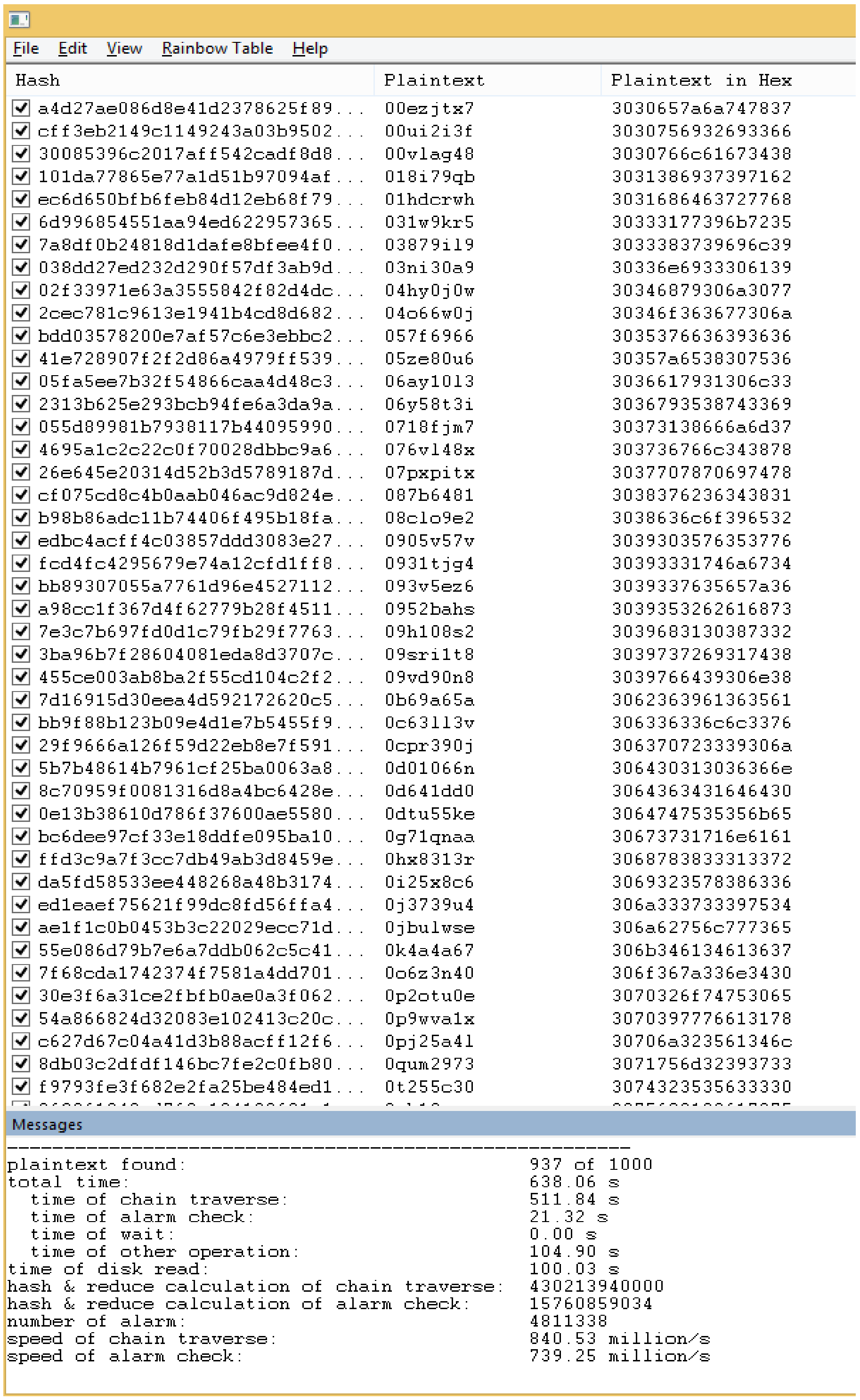
CHARACTER SET DEFINITIONS

Character Set Name	**Character Set Contents**
Numeric	0123456789
Alpha	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Alphanumeric	ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
Loweralpha	Abcdefghijklmnopqrstuvwxyz
Loweralphanumeric	abcdefghijklmnopqrstuvwxyz0123456789
Mixalpha	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
Mixalphanumeric	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
Ascii-32-95	!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~ *
Ascii-32-65-123-4	!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`{|}~ *
Alphanumeric-symbol32-space	ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789!@#$%^&*()-_+=~`[]{}|\:;"'<>,.?/ *

* Represents the space character

Appendix F

RAINBOW TABLE ATTACK SCREEN SHOTS

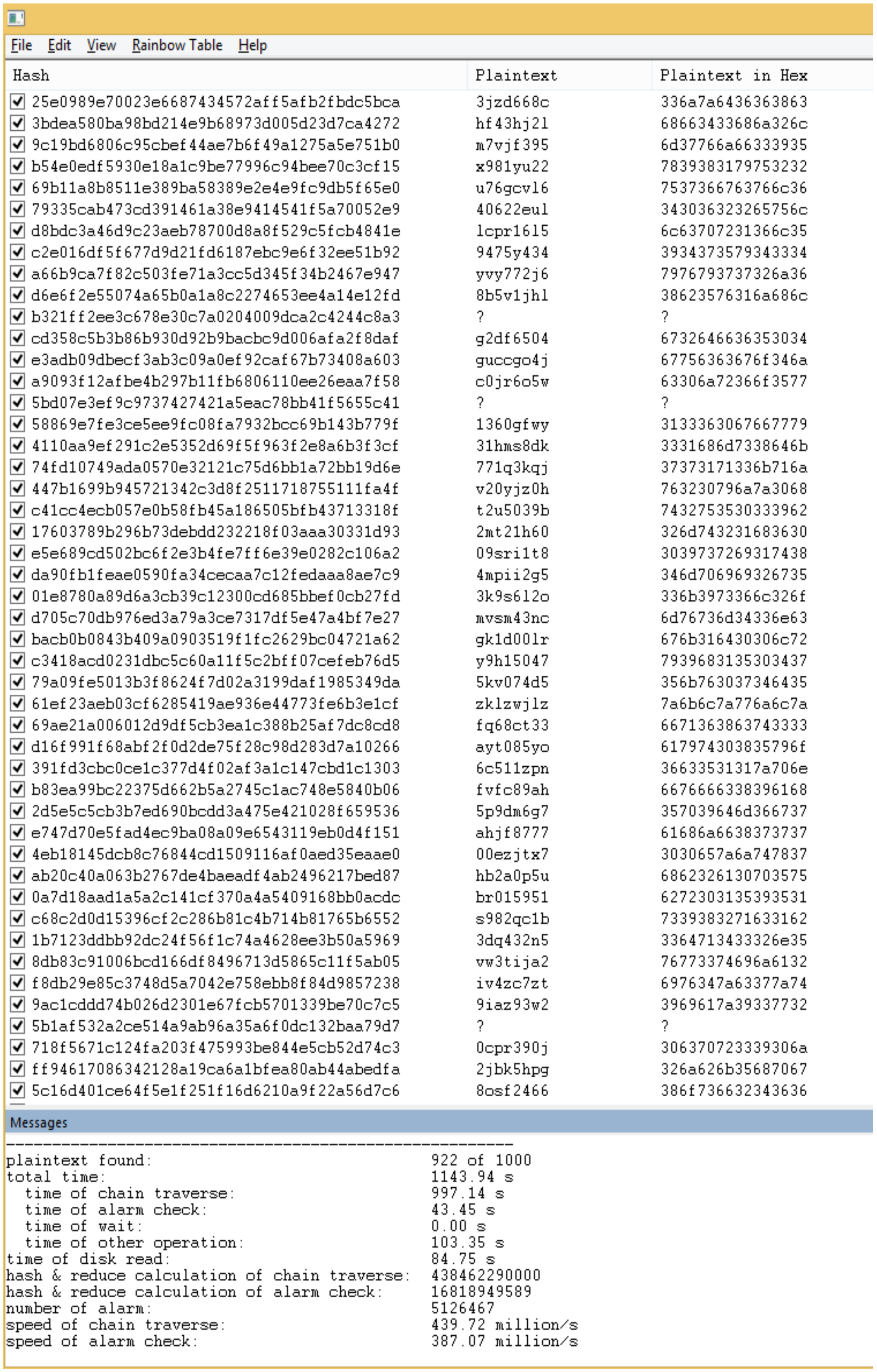


Hash	Plaintext	Plaintext in Hex
✓ a4d27ae086d8e41d2378625f89...	00ezjtx7	3030657a6a747837
✓ cff3eb2149c1149243a03b9502...	00ui2i3f	3030756932693366
✓ 30085396c2017aff542cadf8d8...	00vlag48	3030766c61673438
✓ 101da77865e77a1d51b97094af...	018i79qb	3031386937397162
✓ ec6d650bfb6feb84d12eb68f79...	01hdcrwh	3031686463727768
✓ 6d996854551aa94ed622957365...	031w9kr5	30333177396b7235
✓ 7a8df0b24818d1dafe8bfee4f0...	03879il9	303338373969 6c39
✓ 038dd27ed232d290f57df3ab9d...	03ni30a9	30336e6933306139
✓ 02f33971e63a3555842f82d4dc...	04hy0j0w	30346879306a3077
✓ 2cec781c9613e1941b4cd8d682...	04o66w0j	30346f363677306a
✓ bdd03578200e7af57c6e3ebbc2...	057f6966	3035376636393636
✓ 41e728907f2f2d86a4979ff539...	05ze80u6	30357a6538307536
✓ 05fa5ee7b32f54866caa4d48c3...	06ay1013	30366179313 06c33
✓ 2313b625e293bcb94fe6a3da9a...	06y58t3i	3036793538743369
✓ 055d89981b7938117b44095990...	0718fjm7	303731386 66a6d37
✓ 4695a1c2c22c0f70028dbbc9a6...	076vl48x	303736766c343878
✓ 26e645e20314d52b3d5789187d...	07pxpitx	3037707870697478
✓ cf075cd8c4b0aab046ac9d824e...	087b6481	3038376236343831
✓ b98b86adc11b74406f495b18fa...	08clo9e2	3038636c6f396532
✓ edbc4acff4c03857ddd3083e27...	0905v57v	3039303576353776
✓ fcd4fc4295679e74a12cfd1ff8...	0931tjg4	30393331746a6734
✓ bb89307055a7761d96e4527112...	093v5ez6	303933763565 7a36
✓ a98cc1f367d4f62779b28f4511...	0952bahs	3039353262616873
✓ 7e3c7b697fd0d1c79fb29f7763...	09h108s2	3039683130387332
✓ 3ba96b7f28604081eda8d3707c...	09sri1t8	3039737269317438
✓ 455ce003ab8ba2f55cd104c2f2...	09vd90n8	303976643930 6e38
✓ 7d16915d30eea4d592172620c5...	0b69a65a	3062363961363561
✓ bb9f88b123b09e4d1e7b5455f9...	0c6113v	306336336c6c3376
✓ 29f9666a126f59d22eb8e7f591...	0cpr390j	306370723339306a
✓ 5b7b48614b7961cf25ba0063a8...	0d01066n	306430313036366e
✓ 8c70959f0081316d8a4bc6428e...	0d641dd0	3064363431646430
✓ 0e13b38610d786f37600ae5580...	0dtu55ke	3064747535356b65
✓ bc6dee97cf33e18ddfe095ba10...	0g71qnaa	30673731716e6161
✓ ffd3c9a7f3cc7db49ab3d8459e...	0hx8313r	3068783833313372
✓ da5fd58533ee448268a48b3174...	0i25x8c6	3069323578386336
✓ ed1eaef75621f99dc8fd56ffa4...	0j3739u4	306a333733397534
✓ ae1f1c0b0453b3c22029ecc71d...	0jbulwse	306a62756c777365
✓ 55e086d79b7e6a7ddb062c5c41...	0k4a4a67	306b346134613637
✓ 7f68cda1742374f7581a4dd701...	0o6z3n40	306f367a336e3430
✓ 30e3f6a31ce2fbfb0ae0a3f062...	0p2otu0e	3070326f74753065
✓ 54a866824d32083e102413c20c...	0p9wva1x	3070397776613178
✓ c627d67c04a41d3b88acff12f6...	0pj25a4l	30706a323561346c
✓ 8db03c2dfdf146bc7fe2c0fb80...	0qum2973	3071756d32393733
✓ f9793fe3f682e2fa25be484ed1...	0t255c30	3074323535633330

Messages

```
-------------------------------------------------------
plaintext found:                             937 of 1000
total time:                                  638.06 s
  time of chain traverse:                    511.84 s
  time of alarm check:                       21.32 s
  time of wait:                              0.00 s
  time of other operation:                   104.90 s
time of disk read:                           100.03 s
hash & reduce calculation of chain traverse: 430213940000
hash & reduce calculation of alarm check:    15760859034
number of alarm:                             4811338
speed of chain traverse:                     840.53 million/s
speed of alarm check:                        739.25 million/s
```

Results after searching 1000 MD5 hash values



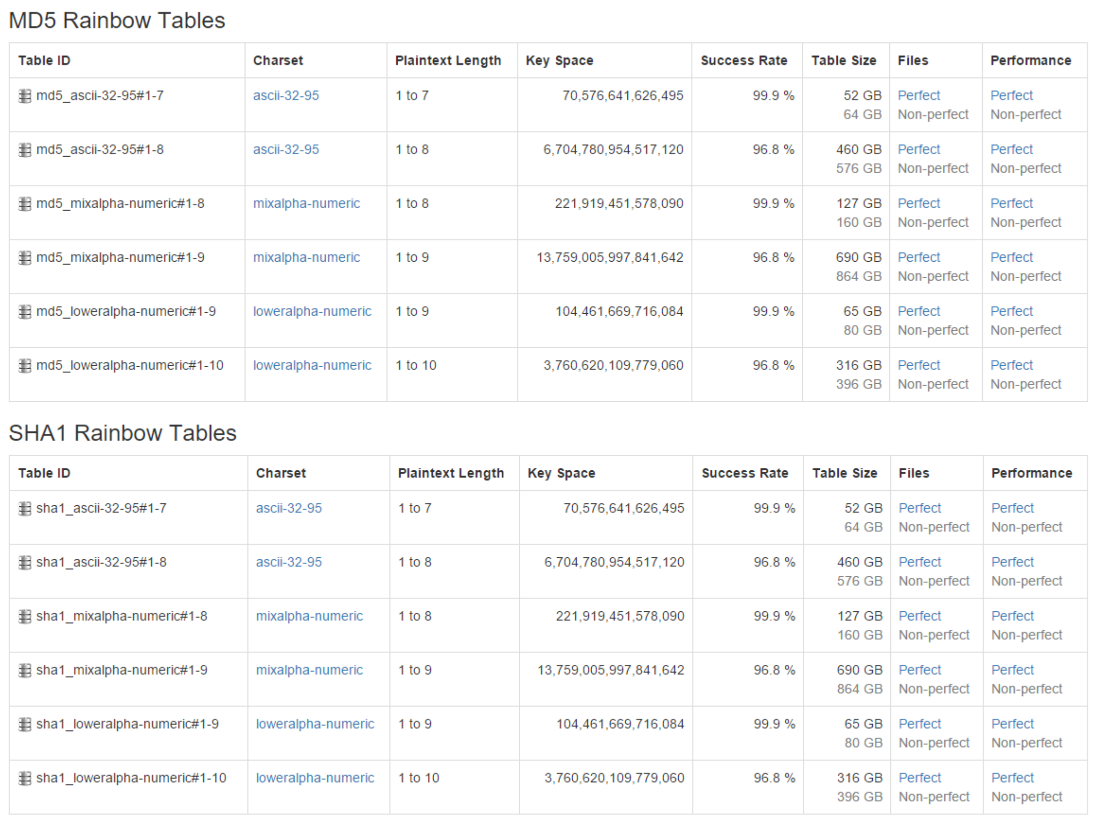
Hash	Plaintext	Plaintext in Hex
✓ 25e0989e70023e6687434572aff5afb2fbdc5bca	3jzd668c	336a7a6436363863
✓ 3bdea580ba98bd214e9b68973d005d23d7ca4272	hf43hj2l	68663433686a326c
✓ 9c19bd6806c95cbef44ae7b6f49a1275a5e751b0	m7vjf395	6d37766a66333935
✓ b54e0edf5930e18a1c9be77996c94bee70c3cf15	x981yu22	7839383179753232
✓ 69b11a8b8511e389ba58389e2e4e9fc9db5f65e0	u76gcvl6	7537366763766c36
✓ 79335cab473cd391461a38e9414541f5a70052e9	40622eul	343036323265756c
✓ d8bdc3a46d9c23aeb78700d8a8f529c5fcb4841e	lcpr1615	6c63707231366c35
✓ c2e016df5f677d9d21fd6187ebc9e6f32ee51b92	9475y434	3934373579343334
✓ a66b9ca7f82c503fe71a3cc5d345f34b2467e947	yvy772j6	7976793737326a36
✓ d6e6f2e55074a65b0a1a8c2274653ee4a14e12fd	8b5v1jhl	38623576316a686c
✓ b321ff2ee3c678e30c7a0204009dca2c4244c8a3	?	?
✓ cd358c5b3b86b930d92b9bacbc9d006afa2f8daf	g2df6504	6732646636353034
✓ e3adb09dbecf3ab3c09a0ef92caf67b73408a603	guccgo4j	67756363676f346a
✓ a9093f12afbe4b297b11fb6806110ee26eaa7f58	c0jr6o5w	63306a72366f3577
✓ 5bd07e3ef9c9737427421a5eac78bb41f5655c41	?	?
✓ 58869e7fe3ce5ee9fc08fa7932bcc69b143b779f	1360gfwy	3133363067667779
✓ 4110aa9ef291c2e5352d69f5f963f2e8a6b3f3cf	31hms8dk	3331686d7338646b
✓ 74fd10749ada0570e32121c75d6bb1a72bb19d6e	771q3kqj	373731713336b716a
✓ 447b1699b945721342c3d8f2511718755111fa4f	v20yjz0h	763230796a7a3068
✓ c41cc4ecb057e0b58fb45a186505bfb43713318f	t2u5039b	7432753530333962
✓ 17603789b296b73debdd232218f03aaa30331d93	2mt21h60	326d743231683630
✓ e5e689cd502bc6f2e3b4fe7ff6e39e0282c106a2	09sri1t8	3039737269317438
✓ da90fb1feae0590fa34cecaa7c12fedaaa8ae7c9	4mpii2g5	346d706969326735
✓ 01e8780a89d6a3cb39c12300cd685bbef0cb27fd	3k9s6l2o	336b3973366c326f
✓ d705c70db976ed3a79a3ce7317df5e47a4bf7e27	mvsm43nc	6d76736d34336e63
✓ bacb0b0843b409a0903519f1fc2629bc04721a62	gk1d00lr	676b316430306c72
✓ c3418acd0231dbc5c60a11f5c2bff07cefeb76d5	y9h15047	7939683135303437
✓ 79a09fe5013b3f8624f7d02a3199daf1985349da	5kv074d5	356b763037346435
✓ 61ef23aeb03cf6285419ae936e44773fe6b3e1cf	zklzwjlz	7a6b6c7a776a6c7a
✓ 69ae21a006012d9df5cb3ea1c388b25af7dc8cd8	fq68ct33	6671363863743333
✓ d16f991f68abf2f0d2de75f28c98d283d7a10266	ayt085yo	617974303835796f
✓ 391fd3cbc0ce1c377d4f02af3a1c147cbd1c1303	6c511zpn	36633531317a706e
✓ b83ea99bc22375d662b5a2745c1ac748e5840b06	fvfc89ah	6676666338396168
✓ 2d5e5c5cb3b7ed690bcdd3a475e421028f659536	5p9dm6g7	357039646d366737
✓ e747d70e5fad4ec9ba08a09e6543119eb0d4f151	ahjf8777	61686a6638373737
✓ 4eb18145dcb8c76844cd1509116af0aed35eaae0	00ezjtx7	3030657a6a747837
✓ ab20c40a063b2767de4baeadf4ab2496217bed87	hb2a0p5u	6862326130703575
✓ 0a7d18aad1a5a2c141cf370a4a5409168bb0acdc	br015951	6272303135393531
✓ c68c2d0d15396cf2c286b81c4b714b81765b6552	s982qc1b	7339383271633162
✓ 1b7123ddbb92dc24f56f1c74a4628ee3b50a5969	3dq432n5	3364713433326e35
✓ 8db83c91006bcd166df8496713d5865c11f5ab05	vw3tija2	7677337469696a6132
✓ f8db29e85c3748d5a7042e758ebb8f84d9857238	iv4zc7zt	6976347a63377a74
✓ 9ac1cddd74b026d2301e67fcb5701339be70c7c5	9iaz93w2	3969617a39337732
✓ 5b1af532a2ce514a9ab96a35a6f0dc132baa79d7	?	?
✓ 718f5671c124fa203f475993be844e5cb52d74c3	0cpr390j	306370723339306a
✓ ff94617086342128a19ca6a1bfea80ab44abedfa	2jbk5hpg	326a626b35687067
✓ 5c16d401ce64f5e1f251f16d6210a9f22a56d7c6	8osf2466	386f736632343636

Messages

```
------------------------------------------------------------
plaintext found:                               922 of 1000
total time:                                    1143.94 s
  time of chain traverse:                      997.14 s
  time of alarm check:                         43.45 s
  time of wait:                                0.00 s
  time of other operation:                     103.35 s
time of disk read:                             84.75 s
hash & reduce calculation of chain traverse:   438462290000
hash & reduce calculation of alarm check:      16818949589
number of alarm:                               5126467
speed of chain traverse:                       439.72 million/s
speed of alarm check:                          387.07 million/s
```

Results after searching 1000 SHA-1 hash values

Appendix G



MD5 Rainbow Tables

Table ID	Charset	Plaintext Length	Key Space	Success Rate	Table Size	Files	Performance
md5_ascii-32-95#1-7	ascii-32-95	1 to 7	70,576,641,626,495	99.9 %	52 GB 64 GB	Perfect Non-perfect	Perfect Non-perfect
md5_ascii-32-95#1-8	ascii-32-95	1 to 8	6,704,780,954,517,120	96.8 %	460 GB 576 GB	Perfect Non-perfect	Perfect Non-perfect
md5_mixalpha-numeric#1-8	mixalpha-numeric	1 to 8	221,919,451,578,090	99.9 %	127 GB 160 GB	Perfect Non-perfect	Perfect Non-perfect
md5_mixalpha-numeric#1-9	mixalpha-numeric	1 to 9	13,759,005,997,841,642	96.8 %	690 GB 864 GB	Perfect Non-perfect	Perfect Non-perfect
md5_loweralpha-numeric#1-9	loweralpha-numeric	1 to 9	104,461,669,716,084	99.9 %	65 GB 80 GB	Perfect Non-perfect	Perfect Non-perfect
md5_loweralpha-numeric#1-10	loweralpha-numeric	1 to 10	3,760,620,109,779,060	96.8 %	316 GB 396 GB	Perfect Non-perfect	Perfect Non-perfect

SHA1 Rainbow Tables

Table ID	Charset	Plaintext Length	Key Space	Success Rate	Table Size	Files	Performance
sha1_ascii-32-95#1-7	ascii-32-95	1 to 7	70,576,641,626,495	99.9 %	52 GB 64 GB	Perfect Non-perfect	Perfect Non-perfect
sha1_ascii-32-95#1-8	ascii-32-95	1 to 8	6,704,780,954,517,120	96.8 %	460 GB 576 GB	Perfect Non-perfect	Perfect Non-perfect
sha1_mixalpha-numeric#1-8	mixalpha-numeric	1 to 8	221,919,451,578,090	99.9 %	127 GB 160 GB	Perfect Non-perfect	Perfect Non-perfect
sha1_mixalpha-numeric#1-9	mixalpha-numeric	1 to 9	13,759,005,997,841,642	96.8 %	690 GB 864 GB	Perfect Non-perfect	Perfect Non-perfect
sha1_loweralpha-numeric#1-9	loweralpha-numeric	1 to 9	104,461,669,716,084	99.9 %	65 GB 80 GB	Perfect Non-perfect	Perfect Non-perfect
sha1_loweralpha-numeric#1-10	loweralpha-numeric	1 to 10	3,760,620,109,779,060	96.8 %	316 GB 396 GB	Perfect Non-perfect	Perfect Non-perfect

Table 3-15 project-rainbowcrack.com MD5 and SHA-1 Rainbow tables for download [68]

References

[1] Coles, Michael, and Rodney Landrum. "Introduction to Encryption." *Expert SQL Server 2008 Encryption*. Apress, 2009. 1-20.

[2] Aravindhan, K., and R. R. Karthiga. "One Time Password: A Survey."*International Journal of Emerging Trends in Engineering and Development*1.3 (2013): 613-623.

[3] Wiedenbeck S., Waters J., Birget J. C., Brodskiy A., and Memon N, 2005, PassPoints: Design and longitudinal evaluation of a graphical password system, International Journal of Human Computer Studies, Vol 63, pp. 102- 127.

[4] Prichard, Janet J., and Frank M. Carrano. *Data Abstraction & Problem Solving with Java: Walls and Mirrors*. 3rd ed. Boston: Addison-Wesley, 2011. Print.

[5] Rouse, Margaret. "What Is Hashing? - Definition from WhatIs.com." *TechTarget*. Whatis.com, 1 Sept. 2005. Web. 4 Jan. 2016. http://searchsqlserver.techtarget.com/definition/hashing.

[6] Katz, Jonathan, and Yehuda Lindell. *Introduction to modern cryptography*. CRC Press, 2014.

[7] András, Keszthelyi. "Life is Short. Have another Affair–Password Security."*Proceedings of FIKUSZ 2015* (2015): 121-130.

[8] "How Rainbow Tables Work." Web. 10 Feb. 2016. http://kestas.kuliukas.com/RainbowTables/ (Very inaccessible and unavailable document to properly credit the author)

[9] Malhotra, Meetika, and Bhushan Dua. "A Review of NTLM Rainbow Table Generation Techniques." *GJCST-E: Network, Web & Security* 13.7 (2013).

[10] Kumar, Naveen. "Password in practice: An usability survey." *Journal of Global Research in Computer Science* 2.5 (2011): 107-112.

[11] Schaffer, Kim. "Are Password Requirements too Difficult?." *Computer* 44.12 (2011): 90-92.

[12] Huth, Alexa, Michael Orlando, and Linda Pesante. "Password security, protection, and management." *United States Computer Emergency Readiness Team* (2012).

[13] Cipresso, Pietro, et al. "How to create memorizable and strong passwords."*Journal of medical Internet research* 14.1 (2012).

[15] Yang, Yulong, Janne Lindqvist, and Antti Oulasvirta. "Text Entry Method Affects Password Security." arXiv Prepr. arXiv1403 (2014).

[16] "The Top 100 Passwords on Ashley Madison." Quartz. Web. 10 Feb. 2016. http://qz.com/501073/the-top-100-passwords-on-ashley-madison/.

[17] Ah Kioon, Mary Cindy, Zhao Shun Wang, and Shubra Deb Das. "Security analysis of MD5 algorithm in password storage." Applied Mechanics and Materials. Vol. 347. 2013.

[18] Aumasson, Jean-Philippe. "Password Hashing: the Future is Now." (2013).

[19] Wang, Xiaoyun, and Hongbo Yu. "How to break MD5 and other hash functions." *Advances in Cryptology–EUROCRYPT 2005*. Springer Berlin Heidelberg, 2005. 19-35.

[20] Sotirov, Alexander, et al. "MD5 considered harmful today, creating a rogue CA certificate." *25th Annual Chaos Communication Congress*. No. EPFL-CONF-164547. 2008.

[21] Joye, Marc, and Francis Olivier. "Side-channel analysis." *Encyclopedia of Cryptography and Security* (2011): 1198-1204.

[22] "Salted Password Hashing - Doing It Right." Secure Salted Password Hashing. Web. 10 Feb. 2016. https://crackstation.net/hashing-security.htm#salt.

[23] "Undergraduate Courses | Computer Science at Virginia Tech." Undergraduate Courses | Computer Science at Virginia Tech. Web. 14 Mar. 2016. http://courses.cs.vt.edu/cs2604/spring04/Notes/C12.Tables.pdf

[24] Keszthelyi, A. (2013). About passwords, ACTA POLYTECHNICA HUNGARICA 10:(6) pp. 99-118., http://www.uni-obuda.hu/journal/Keszthelyi_44.pdf

[25] "Coding Horror." Speed Hashing. 6 Apr. 2012. Web. 10 Feb. 2016. http://blog.codinghorror.com/speed-hashing/.

[26] Ah Kioon, Mary Cindy, Zhao Shun Wang, and Shubra Deb Das. "Security analysis of MD5 algorithm in password storage." *Applied Mechanics and Materials*. Vol. 347. 2013. (THIS TALKS ABOUT RAINBOW TABLES ADD IT TO THE SECTION INSTEAD OF THE OTHER SOURCE)

[27] Forler, Christian, Stefan Lucks, and Jakob Wenzel. "Catena: A Memory-Consuming Password Scrambler." *IACR Cryptology ePrint Archive* 2013 (2013): 525.

[28] Bezzi, Michele; et al. (2011). "Data privacy". In Camenisch, Jan et al. *Privacy and Identity Management for Life*. Springer. pp. 185–186. ISBN 9783642203176.

[29] Aggarwal, Atishay, Pranav Chaphekar, and Rohit Mandrekar. "Cryptanalysis of bcrypt and SHA-512 using Distributed Processing over the Cloud."*International Journal of Computer Applications* 128.16 (2015).

[30] Rivest, R. The MD5 message-digest algorithm. RFC 1321, 37 (April 1992)

[31] Stevens, Marc. "Single-block collision attack on MD5." *IACR Cryptology ePrint Archive* 2012 (2012): 40.

[32] Stevens, M.M.J. “Master’s Thesis On Collisions For MD5.” Eindhoven University Of Technology. June 2007.

[33] Tang, YuChen, Guang Zeng, and WenBao Han. "Classification of disturbance vectors for collision attack in SHA-1." *Science China Information Sciences* (2015): 1-10.

[34] Wang, Xiaoyun, Yiqun Lisa Yin, and Hongbo Yu. "Finding collisions in the full SHA-1." *Advances in Cryptology–CRYPTO 2005*. Springer Berlin Heidelberg, 2005. (Used just for a definition)

[35] "Secure Hashing." NIST.gov. NIST - Computer Security Division - Computer Security Resource Center. Web. 10 Feb. 2016. http://csrc.nist.gov/groups/ST/toolkit/secure_hashing.html

[36] Stevens, Marc. "Cryptanalysis of MD5 & SHA-1." *Proceedings of the Special-Purpose Hardware for Attacking Cryptographic Systems (SHARCS'12)* (2012): 17-18.

[37] Polk, T., et al. *Security Considerations for the SHA-0 and SHA-1 Message-Digest Algorithms*. No. RFC 6194. 2011.

[38] "When Will We See Collisions for SHA-1?" Schneier on Security. Web. 10 Feb. 2016. https://www.schneier.com/blog/archives/2012/10/when_will_we_se.html.

[39] "The SHAppening: Freestart Collisions for SHA-1." The Shappening. Web. 10 Feb. 2016. https://sites.google.com/site/itstheshappening/.

[40] Jose, G. Jai Arul, C. Sajeev, and C. Suyambulingom. "Implementation of data security in cloud computing." *International Journal of P2P Network Trends and Technology* 1.1 (2011): 18-22.

[41] Weir, Matt, et al. "Testing metrics for password creation policies by attacking large sets of revealed passwords." *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010.

[42] Bonneau, Joseph. "The science of guessing: analyzing an anonymized corpus of 70 million passwords." *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012.

[43] "Tips for Creating a Strong Password." Windows.microsoft.com. Microsoft. Web. 10 Feb. 2016. http://windows.microsoft.com/en-ca/windows-vista/tips-for-creating-a-strong-password.

[44] Mirante, Dennis, and Justin Cappos. "Understanding password database compromises." Dept. of Computer Science and Engineering Polytechnic Inst. of NYU, Tech. Rep. TR-CSE-2013-02 (2013).

[45] Graves, Russell Edward. *High Performance Password Cracking by Implementing Rainbow Tables on NVidia Graphics Cards (IseCrack)*. ProQuest, 2008.

[46] Crypto++. Benchmarks-AMD64 Web. 14 March 2016. https://www.cryptopp.com/benchmarks-amd64.html

[47] Kaliski, B. *Password-based cryptography specification*. RFC 2898. Text. 2000. http://tools.ietf.org/html/rfc2898#page-9

[48] Wikipedia. Wikimedia Foundation. Web. 14 Mar. 2016. https://en.wikipedia.org/wiki/Hash_function

[49] "Behind the Scenes of SSL Cryptography." All about SSL Cryptography. Web. 10 Feb. 2016. https://www.digicert.com/ssl-cryptography.htm.

[50] Wiseman, Sarah, et al. "Using checksums to detect number entry error."*Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2013.

[51] "Cryptographic Hash and SHA-3 Standard Development." NIST.gov. NIST. Web. 10 Feb. 2016. http://csrc.nist.gov/groups/ST/hash/.

[52] FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION. "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions." Information Technology Laboratory National Institute of Standards and Technology Gaithersburg, MD 20899-8900, Aug. 2015. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf.

[53] Richard R. Cavanagh, Acting Associate Director for Laboratory Programs. "Announcing Approval of Federal Information Processing Standard (FIPS) 202, SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, and Revision of the Applicability Clause of FIPS 180-4, Secure Hash Standard." The Daily Journal of the United States Government. Web. 5 Aug. 2015. https://www.federalregister.gov/articles/2015/08/05/2015-19181/announcing-approval-of-federal-information-processing-standard-fips-202-sha-3-standard/

[54] Bertoni, Guido, et al. "Keccak." *Advances in Cryptology–EUROCRYPT 2013*. Springer Berlin Heidelberg, 2013. 313-314.

[55] Bertoni, Guido, Joan Daemen, Michaël Peeters, and Gilles Van Assche "The Keccak sponge function family" http://keccak.noekeon.org/.

[56] "Patent US4720860 - Method and apparatus for positively identifying an individual"

[57] Christie, Emma, et al. "Enhanced MIT ID Security via One-Time Passcode." (2015).

[58] "Yubico | Trust the Net with YubiKey Strong Two-Factor Authentication." Yubico. Web. 14 Mar. 2016. https://www.yubico.com

[59] Xie, Tao, and Dengguo Feng. "Construct MD5 Collisions Using Just A Single Block Of Message." *IACR Cryptology ePrint Archive* 2010 (2010): 643.

[60] "Patching the Perpetual MD5 Vulnerability." Venafi. GAVIN HILL, DIRECTOR, PRODUCT MARKETING AND THREAT INTELLIGENCE. Web. 10 Feb. 2016. https://www.venafi.com/blog/post/patching-the-perpetual-md5-vulnerability/.

[61] "GOOGLE ATAP." Project Abacus ~. Web. 10 Feb. 2016. http://googleatap.blogspot.ca/2015/11/project-abacus.html.

[62] Google Developers (29 May 2015). *Google I/O 2015 – A little badass. Beautiful. Tech and human. Work and love.* [Video file]. Retrieved from https://www.youtube.com/watch?v=mpbWQbkl8_g

[63] Zeifman, Igal. "Deprecating SHA-1 — Why, What and When." Imperva Incapsula. 21 Oct. 2014. Web. 10 Feb. 2016. https://www.incapsula.com/blog/sha-1-hash-algorithm-deprecation.html.

[64] Nuwer, Rachel. "Wristband unlocks your devices with your heartbeat." *New Scientist* 219.2933 (2013): 19.

[65] Nymi Inc. "Nymi." Nymi.com. Web. 10 Feb. 2016.

https://www.nymi.com/wp-content/uploads/2013/11/NymiWhitePaper-1.pdf.

[66] "Nymi Inc." Nymi. Web. 14 Mar. 2016. https://www.nymi.com/

[67] Hellman, Martin E. "A cryptanalytic time-memory trade-off." *Information Theory, IEEE Transactions on* 26.4 (1980): 401-406.

[68] "List of Rainbow Tables." List of Rainbow Tables. Web. 14 Mar. 2016. http://project-rainbowcrack.com/table.htm

[69] Brown, Kelly. "The dangers of weak hashes." *SANS Institute InfoSec Reading Room, Tech. Rep* (2013).

[70] "FreeRainbowTables.com." FreeRainbowTables.com. Web. 14 Mar. 2016. https://www.freerainbowtables.com/en/tables2/