IB Computer Science
Extended Essay

Investigation Into 256-bit Hashing and Its Vulnerability to Increasing Computing Speeds.

To What Extent Will Advances in Computing Speeds Negate the Security Provided by 256-bit Password Hashing?

May 13th, 2018
Word Count: 3625

Contents

Terminology

Cryptographic Hash Function (krɪptəˈgræfɪk hæʃ ˈfʌnŋkʃən) *Noun* • A mathematical function used to generate a pseudo-random output (image) of a fixed length from an input (preimage) of any given length. [23]

Decoding (ˈdiːkəʊdɪŋ) *Verb* • The use of an inverse function in order to option the preimage of an encryption. [16]

Encode (ɪnˈkəʊd) *Verb* • To format data into a standard format that transforms its meaning. [16]

Encryption (ɪnˈkrɪpʃ(ə)n) *Verb* • The use of a function to format plaintext data (preimage) and encode it in a way that renders it not useable in the same way as its plaintext was intended to be used – usually the image is not human-readable and must be decoded before use. [16]

Hacker (ˈhækə) *Noun* • Someone who undertakes the act of hacking with the malicious intent – in this case to obtain a password protecting secure data.

Hacking (ˈhækɪŋ) *Verb* • In our case, hacking refers to the use of attacks on a hash (image) in order to obtain its preimage.

[Computed] Hash (hæʃ) *Noun* • The output (image) of a cryptographic hash-function, of a fixed length depending on the function used. The image of the function. [17]

Hashing (ˈhæʃɪŋ) *Verb* • The act of producing an image from a plaintext input (preimage) in order to obfuscate the input – in this essay passwords are hashed using the SHA-256 hash-function. [17]

Image (ˈɪmɪʤ) *Noun* • The output of a function, hash-function, encryption or otherwise. [21]

[Password] Security (ˈpɑːswɜːd) *Noun* • The hiding of private information such as banking details behind a string allowing access to the information – in this case stored as a hash.

PreImage (priː-ˈɪmɪʤ) *Noun* • The input to a function, hash-function, encryption or otherwise. [21]

Note

[x] = Reference to bibliographic index.
^{x} = Reference footnote index.

1 Introduction

1.1 Background

The security of private data on computer systems relies on the use of cryptography. Encryption and hashing are two key aspects that make up cryptography over the internet as we know it today, these are methods of converting a set of data (plaintext) into an encoded version of that text; in order to obfuscate the original data. The data is encrypted by means of an pseudo-random algorithm [30].

The encoded version of the data is almost useless unless you have the technology to find out what the original text was. This method is used for passwords – the passwords stored by websites are not stored in their original form, instead they are first encoded so that if the database the password is stored in is breached, it is challenging to decode the text to find what the original password was.

This essay will focus on *cryptographic hash functions* as opposed to other forms of cryptography, as hash-functions are the industry standard for storing passwords. [27] We use hash functions because the images they produce are said to be impossible to reverse and find the preimage of [21]. The implication of this is that data can be encoded and only decoded if the original data is presented. Because of this, websites can store the hashes without fear that hackers may be able to access users' passwords, even if the hashes are found. Though, there are ways hackers can find the preimage of hashes if users are not careful with passwords they choose.

Hash functions are deterministic formulae, producing the same output with a given input each time [19]. For this essay I will be exploring *SHA-256*, a type of hash function, as it is the industry standard. [19] [27] We use this function for the following reasons: it produces a long 256-bit hash (a series of 256 `1`s and `0`s); it is yet to be broken by means of working backwards through the function in order to find the preimage of a given image; [19] and the long image length lends itself to being preimage resistant, second-preimage resistant and collision resistant [21] - each explained in **subsection 2.2**.

I believe that security of private data is vitally important to our lives, and thus I will be investigating my research question;

”To what extent will advances in computing speeds negate the security provided by 256-bit password hashing?”

The relevance of this topic is that as computing speeds increase it is ever-easier for hackers to use the methods I will outline later in order to circumvent security. The outcome of my research will determine whether it is necessary to upgrade security protocols to something even more secure than SHA-256, or if data is safe with the current standards in place.

I will first look at hash functions and what makes them secure, then look at the kinds of attacks that can be used on hashes in order to obtain their preimage. Thenceforth I will look at ways companies can further secure hashes using additional technologies such as ’salts’ and ’peppers’. Finally, I will discuss the feasibility of successfully obtaining the preimage of a hash once the necessary precautions are taken to protect it.

1.2 Historical Analysis

In the past, encryption was used to secure passwords and other secret messages. These methods have been since deemed insecure [12] [24]. When the use of hash functions became standard, even early hashes were not foolproof, experiencing insecurities due to their short length of hash they produced:

Before hashes, ciphers such as 1970's **Data Encryption Standard** (DES) developed by IBM, required keys in order to 'unlock' the data. [12] This particular standard of encryption became obsolete once it was determined that computing speeds were fast enough to generate sufficient attempts at unlocking the cipher through a trial-and-error, inputting *56-bit key*, after two companies banded together in 1999 to break a DES cipher, taking only 22 hours. [12] 56 bits is a relatively short length of bits when compared to a modern encryption standard. To give a sense of scope, a 56-bit number has $6.22 * 10^{59}$ [1] times fewer permutations than a 256-bit number, and $2.11 * 10^{20}$ [2] times fewer permutation than even a 128-bit number, going to show just how much more computer processing time it would take to trial-and-error through these larger bit-count hashes [or encryptions].

Furthermore, in the more recent past **SHA-1**, was used as a standard for hashing. [23] Published in 1995, preceding SHA-2 (current), SHA-1 used a *160-bit* model, generating hashes of that length. [24] Like SHA-2, this function is deterministic, producing a 160-bit image for a plaintext preimage. Like DES, trial-and-error can be used. [24] [23] Plaintext preimages are to be fed into the function until an image matches the hash you are trying to break. The determination that 160-bit was too short a hash length in 2005 lead to 256-bit encryptions becoming the norm, due to its relative size and time it would take to iterate through so many permutations of images, [15] ensuring an increase in the security in common hash functions. [24] [23] [20]

[1] $2^{56}/2^{256} * 100\% = 6.22e59\%$
[2] $2^{56}/2^{128} * 100\% = 2.11e20\%$

2 Investigating Hash Encryptions

2.1 Hashes in General

Example of a 256-bit hash on two sample strings.

String input (Hash Function's Preimage):

"Password"

Output of hash function (256 bits):

1011110100010000100100010011000110110100010100000000010000000
000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000
000000000000000

Data stored in database (Hexadecimal equivalent of the above binary):

5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8

On a very similar string input...

String input:

"Dassword"

Output of hash function:

101010111101100011000001011100110111010011110111001010000000
000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000
000000000000000

Data stored in database:

55ec60b9ba7b924bf480439fd5473e00325f71f5454403d3d2705ddd152ac9cf

Figure 1.1: Showing the different hashes of two similar preimages, where the outputs are vastly differing.

In terms of website security, the figure 1.1 displays how two passwords look once passed through a hash-function. The plaintext password ’Password’ is not stored in a website’s database, but its corresponding hash is. Any minor change to the plaintext will completely change what the hash looks like. So, ’Password’ and ’Dassword’ have completely different outputs. Note that the output is the same length for each.

So when a user goes to log back into their account, the password they enter is hashed and the output checked against the hash on file for that account - if they match the user may log in.

Almost all modern programming languages now have in-built encryption packages. For example, below is the body of a Java method, converting the input ”Password” into its 64-bit hash. [28] This makes the ability to secure information open for anyone to use.

```
// Base64Encryption.java
...
    String inputString = "Password"; //Input

    //Instance of Java class containing hash function.
    MessageDigest sha256 = MessageDigest.getInstance("SHA-256");

    //Convert string to byte array, then hash with #digest() method.
    byte[] inputArray = inputString.getBytes();
    byte[] hashArray = sha256.digest(passBytes);

    ... //Code reconstructing hash as a string from a byte array.

    //Output: 5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8
    System.out.println(hashArrayAsString);
...
```

Listing 1.1: Code displaying simplicity of hashing in a modern language.

2.2 Attacks on Hashes

2.2.1 Preimage Attacks

In cryptographic terms, preimage attacks are attacks on a hash-functions' preimage, the inverse of the product of a function. e.g, for $f(x) = x^2$, where the product is 4, the inverse images are a set $\{-2, 2\}$). [21] In a preimage attack, the focus is placed on finding said preimage.

Hash-functions attempt to use two forms of security against attacks: Preimage resistance and second-preimage resistance. [21] [20] The former is ensuring it is computationally infeasible to find the preimage from the output of a function through brute-force (see below), and the latter is that it is computationally infeasible to find a second preimage that yields a like image. (i.e, given that $x' \neq x$, it is difficult to find $h(x) = h(x')$)[21].

We are yet to mathematically find the inverse of the SHA-256 hash-function, and so we can say it is 'one way'. [15] Therefore, the only way we can find a specific preimage is by inserting a variety of inputs into the hash function and guess-and-checking the output until it matches the hash you are attempting to know the preimage of. We can call this the method 'verification' for short.

Practically, a hacker may access a database of passwords stored as hashes, and run a program that iterates through many SHA-256 attempts with a range of preimages until it finds an output that is the same as an image stored in the database, at which point the hacker knows what the password for that user's account is (the preimage).

To summarise: a preimage attack can be used when a computed hash is known and the input of the hash-function used to produce said hash is desired.

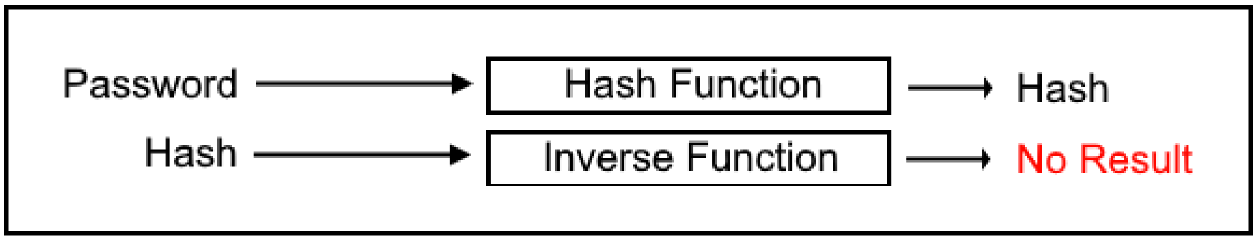

Figure 2.1: Illustration of the one-way nature of hash functions.

Knowing this, there are a few ways one may attempt to obtain the password corresponding to a hash. Some methods include:

Brute-force

The slowest and least efficient method of finding the corresponding preimage of a hash is brute-force. Brute-force methods do as the name suggests, attempting every possible input into a hash-function incrementally until the correct input is found (through the aforementioned verification approach). [18] Due to the fixed-length of the function's range, there are 2^{256} permutations of possible outputs, [15] and on average the correct image is found after 2^{255} attempts (50%of the total permutations), therefore 2^{255} attempts are, on average, necessary to find the preimage.

Rainbow tables

A rainbow table is a list of preimages with their corresponding images [8] - someone who has a hash and wants to know the preimage required to output it can look up the hash in this table in order to find its preimage. This is only likely to be successful if the hash is for a commonly used password (like names of places, dates, names of people or 'password'). [8] Some tables may take into account varying versions of common passwords, like when numbers are appended - however it is infeasible for a table to contain every combination of hash and input. [11] This method sacrifices local storage space for reduced processing time (Lookups are a faster computational operation than completing a full hash). [8] One more drawback of this method is that if the hash-function used a *'salt'* or *'pepper'*, the hashes in the table are useless. Salts and peppers are explained in **subsection 2.3**. [11] [5]

Password (preimage)	Hash (image, as hexadecimal)
Password123	008c70392e3abfbd0fa4 . . .
London	ecc0e7dc084f141b2947 . . .
ILoveDogs	bd22742a359acb57af42 . . .
. . .	. . .

Table 1.1: A portion of a Rainbow Table - preimage and a portion of their SHA-256 image.

Dictionary attacks

Dictionary attacks are fundamentally similar to rainbow tables, however they do not perform lookups, instead they strategically choose which preimages to verify based off of large banks of words, phrases, names and common passwords (Such as a dictionary). [6] Iterating through these possibilities skips all of the very unlikely preimages that the brute-force method would run through, instead focusing on those a user is likely to be able to remember and use, creating variations like substituting letters for numbers and verying capitalisation as a human might. [6] However, even this attack can be thwarted through the use of *’salts’* and *’peppers’*. [5] [6]

2.3 Providing Extra Security to Hashes

The attacks listed in the previous section – aside from bute-force – rely on predictable passwords. Without extra security, a rainbow table lookup can find hashes corresponding to simple passwords like ’cat’ or ’Awesome123’ or ’Password’ in seconds. But, companies can take extra measures when storing passwords.

Salts

Salts are a pseudo-randomly generated string of data that is generated when a user submits their password to be hashed and stored by a system. Before hashing, the system would concatenate the ’salt’, a newly generated string, to the end of the user’s password, then produce the computed hash. [11] The computed hash is then stored alongside the salt in the database, so the next time a user looks to login, the salt is reapplied before the verification of the hash takes place. This protects against pre-computed rainbow table attacks[5], as the hash of like passwords are likely very different due to the applied salt, but does not protect against dictionary attacks, as if the attacker knows the hash, they know the salt and can apply it before each iteration. [11]

Peppers

Peppers are similar to salts in that they are concatenated to the password before encryption, however, they are not stored in the database alongside the computed hash instead they are kept as a secret by the client, used as a constant across all encrypted passwords. [13] These are kept secure, away from the database so that the attacker would need both the pepper and the database (salt and hash) in order to apply a dictionary attack (making them more infeasible). [13] [11] [5]

If companies apply both salts and peppers to the passwords that are made, the company is now covered on almost all bases from different forms of attack. This leads us to investigate brute-force as the last method.

2.4 Expression of the Brute-Force Method

Assuming a company applies salts and peppers to hashes, and a hacker is unable to use the faster methods of validating a hash they might choose to brute-force it.

Deterministic hash-functions can be illustrated as the following function literal, where m, is the input to a hash-function h, $m \subset Z$, and 256 is the number of bits in the output computed hash [20]. [3]...

$$\begin{aligned} h : m &\longrightarrow 2^{256} \\ &\Rightarrow y, \text{of length 256 bits.} \\ &= h(m) \end{aligned} \tag{1}$$

A brute force attack iterates through permutations of domain m in order to determine outputs of y that match the hash they are verifying. When $y_{output} = y_{current}$. Iteration i for $i = 1, 2, \ldots, 2^n$. [Author Owned]

$$\begin{aligned} h(m_i) &= y_{current} \\ &\Rightarrow y_{output} \end{aligned} \tag{2}$$

[3]Hash functions are of algorithmic efficiency $O(2^n)$.

Or computationally [Author Owned]...

```
m = 0
found = false
y = CURRENT_HASH

loop while found is false
    x = m.hash()
    if x = y
        set found to true
        output m
    else
        increment m
    endif
end loop
```

Listing 2.1: Pseudocode showing a simple brute-froce algorihm.

When trying to find the *correct* value of m, brute-force attempts every permutation of x (in the above listing) and thus has the highest chance of finding m. [Author Owned]

2.5 Feasibility of Cracking 256-bit Encryptions

We can look at how feasible cracking a 256-bit hash is from two perspectives: time and energy. If computers were capable of computing enough hashes to make brute forcing one viable, how much speed and power would they require?

We will be looking at GPUs (graphics cards / graphics processing units) as the computer component that deals with computing the hashes as they are better suited for being used in large arrays and engaging in parallel processing than CPUs. Let us assume that each is running at a speed equal to their maximum potential. Speed will be measured in FLOPS (floating point operations per second), as a unit of computational speed. A GFLOPS (gigaFLOPS) is 10^9 FLOPS.

2.5.1 Time

In a computer targeted towards generating hashes, we can assume the specific graphics card being used by looking at what bitcoin and other cryptocurrency miners use [4]. Below is a table showing the highest selling GPUs to miners, and their corresponding speeds. [29]

	Specifications	
GPU Model	Speed (GFLOPS)	Release Date
Radeon RX 470 [2]	4367	Aug 2016
Radeon RX 480 [3]	5498	Jun 2016
Radeon R9 295X2 [4]	5733	Apr 2014
Radeon HD 7990 [1]	8200	Apr 2013

Table 2.1: Graphics cards most commonly used in bitcoin mining (repetetive hashing)

The Radeon RX 480 is a mid-tier average card that is setup in large arrays for hashing SHA-256 over and over again to generate bitcoins, so we will use that card's speed in our speed calculations - 5498 FLOPS - which is about 5498 billion calculations per second. For this example let's say we have one thousand of these GPUs in a parallel computing

[4]Bitcoin mining consists of generating hashes until you find a hash that acts as a key in order to generate a bitcoin - they use the same brute-force process as hacking.

system, each performing at maximum efficiency.

$$1000 * 5,498,000,000,000 = 5498 \text{ TFLOPS (Trillion FLOPS)}$$

In one day (86400 seconds) the GPU can do $475*10^{19}$ computations ($86400*5498$ TFLOPS). If we divide the number of average permutations taken to find a hash by the number of computations per day we can make, we will find how many days it will take to find the correct preimage for the given hash.

$$\frac{2^{255}}{475*10^{19}} = 1.219*10^{56} \text{ days}$$
$$\frac{1.219*10^{56}}{365} = 3.339*10^{53} \text{ years}$$

Given that the estimated age of the universe is 13.799 ± 0.021 billion years old, we could say that with one thousand of these graphics cards running all day every day, it would take as long as the universe has existed times by $2.4*10^{43}$ in order to find the preimage of a single hash through brute force. With the same calculations, if you had one billion GPUs in parallel, it would take $3.339*10^{47}$ years to find the preimage.

This truly gives a sense of scale of just how large a 256-bit number is. [5] What about the *fastest* computer in the world? At 124.5 petaFLOPS being its theoretical maximum top speed, the Sunway TaihuLight [10] supercomputer would still take $1.474*10^{52}$ years to crack the hash. It is clear through these calculations that as it stands, in the foreseeable future, we will not have computers fast enough to iterate through 2^{255} permutations of a binary number.

[5] $2^{256} = 115,792,089,237,316,195,423,570,985,008,687,907,853,269,984,665,640,564,039,457,584,007,913,129,639,936$

2.5.2 Energy

To look at the lowest possible energy consumption that would be needed to iterate through 2^{255} permutations of 1s and 0s, we look towards Landauer's Principle. This principle finds the theoretical minimum energy consumption for the transformation, known as the Landauer Limit, of a single bit of data using the following equation: [9]

$$kT \ln 2$$

Where k is a the Boltzmann Constant, $1.381023 J/K$ (joules per kelvin), T is the environmental temperature of the system, and $\ln 2$ is a constant being the natural logarithm of 2, ≈ 0.69315.

Let us assume we are computing in as cold an environment we can achieve without expending energy, the ambient temperature of space, which is $3.2K$ (Kelvin) [26] [22]. The lower limit of energy required to change a bit according to Landauer's Principle is:

$$(1.38 * 10^{23})(3.2)(\ln 2) = 3.061 * 10^{-23} J$$

For each permutation of the 256-bit number, we must change a single binary digit to a 1. The addition of a bit has an energy cost associated, however the removal of a bit (1 to 0) does not. [9] As we are changing a single 0 to a 1 for each permutation, we can multiply the energy cost by the number of permutations to calculate the energy cost to iterate through 2^{255} permutations of the image.

$$3.061 * 10^{-23} * 2^{255} = 1.772 * 10^{54} J$$

A display of how one bit is added for each permutation is displayed below in table 2.2.

Iteration	Permutation
1	…0001
2	…0010
3	…0011
4	…0100
5	…0101
…	…

Table 2.2: Example of permutations of hashes adding a single bit each iteration.

The amount of energy required ($1.772 * 10^{54} J$) is extreme. As a comparison, the yearly output of energy of the sun is only $1.245 * 10^{11} J$. [14]

3 Implications of Increasing Computing Speeds

When you consider the orders of magnitude more speed and power that is required in order to find the preimage of a hash through brute force, it is logical to think that with current technological limitations presented by silicon-based computers and our current means of producing energy, it is very unlikely that we will be able to achieve such a task any time soon.

Even when measuring energy consumption in the most optimal conditions – in the vacuum of space, with GPUs running at no inefficiency – we would require millions of years and the ability to harness masses of solar energy in order to obtain such power.

As computer speeds increase, unless new technology is discovered that surpasses current limitations, we are safe from computers' abilities to crack 256-bit hashes through brute force. With the rise of quantum computing and other experimental technologies, we do however come closer to a reality where passwords may not be safe, but this is experimental and not yet within the scope of investigation. [7]

4 Conclusion

4.1 Propositions

The realistic methods hackers may use: rainbow tables; dictionary attacks; and social engineering (obtaining the password through non-cryptographic means), can be curbed when companies take precautions to secure their databases containing hashes and when they use appropriate technologies such as salts and peppers. Companies can impose restrictions on the types of passwords users can choose such that they must include symbols, capitalisation and other variations in order to further protect against non-brute-force methods. [5]

4.2 Closing Statements

To answer my research question: ”To What Extent Will Advances in Computing Speeds Negate the Security Provided by 256-bit Password Hashing?”

The increases in computing speeds in computers as we know them will have little to no effect on the security provided by hash algorithms. The security 256-bit hashes provide is that they are very long in length, and through this they repel attacks (brute force) that make them fundamentally insecure. They are also complicated enough in nature that they can not be mathematically inverted. They do not, however, repel dictionary and rainbow table attacks, but this is for users and companies to work against, whilst SHA-256 and other 256-bit hashes do as they intend.

Throughout this essay, academic papers and university websites were key sources in order to limit the chance of citing inaccurate information. Each source was evaluated for their academic merit based off of the credentials of the author and the peer checking involved. How this essay has concluded, if any miss-information was used it is unlikely to change the essay’s conclusion or outcome.

Works Cited

[1] AMD. (2011, September 10). AMD Radeon HD 7990 Graphics Card. Retrieved May 10, 2018, from https://www.amd.com/en-us/products/graphics/desktop/7000/7990

[2] AMD. (2016, July 29). Radeon RX 470 Graphics Cards. Retrieved May 10, 2018, from https://www.amd.com/en-us/products/graphics/radeon-rx-series/radeon-rx-470

[3] AMD. (2016, June 29). Radeon RX 480 Graphics Card. Retrieved May 10, 2018, from https://www.amd.com/en-us/products/graphics/radeon-rx-series/radeon-rx-480

[4] AMD. (2017, July 31). AMD Radeon R9 Series Gaming Graphics Cards with High-Bandwidth Memory. Retrieved May 10, 2018, from https://www.amd.com/en-us/products/graphics/desktop/r9#

[5] Alexander, S. (2012, June 20). *Passwords Matter*. Retrieved March 10, 2018, from http://bugcharmer.blogspot.sg/2012/06/passwords-matter.html

[6] Atwood, J. (2009, January 7). Dictionary Attacks 101. Retrieved March 20, 2018, from https://blog.codinghorror.com/dictionary-attacks-101/

[7] Bernstein, D. (2009). Introduction to post-quantum cryptography. Retrieved March 29, 2018, from http://www.pqcrypto.org/www.springer.com/cda/content/document/cda_downloaddocument

[8] Boneh, D. (2003). Advances in cryptology - CRYPTO 2003. Berlin: Springer.

[9] Bennett, C. (2003). *Notes on Landauer's principle, reversible computation, and Maxwell's Demon*. IBM Research Division, Yorktown Height, New York.

[10] Clark, J., & King, I. (2016, June 20). *World's Fastest Supercomputer Now Has Chinese Chip Technology*. Retrieved May 2, 2018, from https://www.bloomberg.com/news/articles/2016-06-20/world-s-fastest-supercomputer-now-has-chinese-chip-technology

[11] Crackstation. (2017, August 1). Salted Password Hashing - Doing it Right. Retrieved March 20, 2018, from https://crackstation.net/hashing-security.htm#salt

[12] Diffie, W., & Hellman, M. E. (1977, June). *Exhaustive Cryptanalysis of the NBS Data Encryption Standard*. Retrieved March 27, 2018, from https://web.archive.org/web/20140226205104/http://origin-www.computer.org/csdl/mags/co/1977/06/01646525.pdf

[13] Forler, C., Lucks, S., & Wenzel, J. (2013). Catena: A memory-consuming password-scrambling framework. Cryptology ePrint Archive, Report 2013/525.

[14] Institute of Agriculture, The University of Tennessee. (2013, January 12). The Sun's Energy. Retrieved May 10, 2018, from https://ag.tennessee.edu/solar/Pages/What Is Solar Energy/Sun's Energy.aspx

[15] Lamberger, M., & Mendel, F. (2011). *Higher-Order Differential Attack on Reduced SHA-256*. Retrieved March 28, 2018, from https://eprint.iacr.org/2011/037.pdf

[16] Miessler, D. (2011, May 28). *Encoding vs. Encryption vs. Hashing vs. Obfuscation*. Retrieved May 11, 2018, from https://danielmiessler.com/study/encoding-encryption-hashing-obfuscation/

[17] Northcutt, S. (2008, January 10). *Hash Functions*. Retrieved May 11, 2018, from https://www.sans.edu/cyber-research/security-laboratory/article/hash-functions

[18] Paar, C., & Pelzl, J. (2010). *Understanding cryptography: A textbook for students and practitioners*. Berlin: Springer.

[19] Penard, W., & Van Werkhoven, T. (2014, October 14). *On the Secure Hash Algorithm family*. Retrieved March 21, 2018, from https://web.archive.org/web/20160330153520/http://www.staff.science.uu.nl/~werkh108/docs/study/Y5_07_08/infocry/project/Cryp08.pdf

[20] Potter, H. (2016, April 25). *Hash function*. Retrieved March 09, 2018, from https://www.slideshare.net/HarryPotter40/hash-function-61328365

[21] Rogaway, P., & Shrimpton, T. (2004, February 12). *Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Restistance, and Collision Resistance*. Retrieved March 10, 2018, from http://web.cs.ucdavis.edu/~rogaway/papers/relates.pdf

[22] Schneier, B. (2015). *Applied cryptography: protocols, algorithms, and source code in C* (2nd ed., Vol. 1). Indianapolis, IN: Wiley.

[23] Schneier, B. (2004, August 19). *Cryptanalysis of MD5 and SHA: Time for a New Standard*. Retrieved May 11, 2018, from https://www.schneier.com/essays/archives/2004/08/cryptanalysis_of_md5.html

[24] Schneier, B. (2005, February 19). *Cryptanalysis of SHA-1*. Retrieved March 27, 2018, from https://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html

[25] SHA-2. (2018), Retrieved April 26, 2018, from https://en.wikipedia.org/wiki/SHA-2#Pseudocode, Note: Wikipedia being the **primary** source of this information. Author originally uploaded to Wikipedia.

[26] Space.com. (2012, February 29). *What's the Temperature of Outer Space?* Retrieved May 10, 2018, from https://www.space.com/14719-spacekids-temperature-outer-space.html

[27] Stevens, J., & Righetto, D. (2018, March 18). *Password Storage Cheat Sheet*. In J. Manico (Ed.), from https://www.owasp.org/index.php/Password_Storage_Cheat_Sheet.

[28] Thorin86. (2014, August 21). *SHA2 password hashing in java*. Retrieved January 08, 2018, from https://stackoverflow.com/questions/6840206/sha2-password-hashing-in-java

[29] Tuwiner, J. (2017, July 13). Ethereum Mining Hardware. Retrieved May 10, 2018, from https://www.buybitcoinworldwide.com/ethereum/mining-hardware/

[30] US. Office for Civil Rights. (2013, July 26). *What is encryption?*. Retrieved January 08, 2018, from https://www.hhs.gov/hipaa/for-professionals/faq/2021/what-is-encryption/index.html

Appendices

Appendix A Source pseudocode for the SHA-256 algorithm [25]:

```
Lemma 1: All variables are 32 bit unsigned integers and addition is
              calculated modulo 232
Lemma 2: For each round, there is one round constant k[i] and one entry
              in the message schedule array w[i], 0 i 63
Lemma 3: The compression function uses 8 working variables, a through h
Lemma 4: Big-endian convention is used when expressing the constants
              in this pseudocode, and when parsing message block data from
              bytes to words, for example, the first word of the input message
              "abc" after padding is 0x61626380

#Initialize hash values:
#(first 32 bits of the fractional parts of the square roots of
#the first 8 primes 2..19):
h0 := 0x6a09e667
h1 := 0xbb67ae85
h2 := 0x3c6ef372
h3 := 0xa54ff53a
h4 := 0x510e527f
h5 := 0x9b05688c
h6 := 0x1f83d9ab
h7 := 0x5be0cd19

#Initialize array of round constants:
#(first 32 bits of the fractional parts of the cube
#roots of the first 64 primes 2..311):
k[0..63] :=
   0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5,
     0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
   0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3,
     0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
   0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc,
     0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
   0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7,
     0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
   0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13,
     0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
   0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3,
     0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
   0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5,
     0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
   0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208,
     0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2
```

```

Pre-processing (Padding):
begin with the original message of length L bits
append a single '1' bit
append K '0' bits, where K is the minimum number >= 0 such that
          L + 1 + K + 64 is a multiple of 512
append L as a 64-bit big-endian integer, making the total post-
          processed length a multiple of 512 bits

Process the message in successive 512-bit chunks:
break message into 512-bit chunks
for each chunk
   create a 64-entry message schedule array w[0..63] of 32-bit words
   copy chunk into first 16 words w[0..15] of the message schedule array

   Extend the first 16 words into the remaining 48 words w[16..63] of the
          message schedule array:
   for i from 16 to 63
      s0 := (w[i-15] rightrotate 7) xor (w[i-15] rightrotate 18)
                    or (w[i-15] rightshift 3)
      s1 := (w[i-2] rightrotate 17) xor (w[i-2] rightrotate 19)
                    xor (w[i-2] rightshift 10)
      w[i] := w[i-16] + s0 + w[i-7] + s1

   Initialize working variables to current hash value:
   a := h0
   b := h1
   c := h2
   d := h3
   e := h4
   f := h5
   g := h6
   h := h7

   Compression function main loop:
   for i from 0 to 63
      S1 := (e rightrotate 6) xor (e rightrotate 11) xor (e rightrotate 25)
      ch := (e and f) xor ((not e) and g)
      temp1 := h + S1 + ch + k[i] + w[i]
      S0 := (a rightrotate 2) xor (a rightrotate 13) xor (a rightrotate 22)
      maj := (a and b) xor (a and c) xor (b and c)
      temp2 := S0 + maj

      h := g
      g := f
      f := e
      e := d + temp1
```

```
        d := c
        c := b
        b := a
        a := temp1 + temp2

    Add the compressed chunk to the current hash value:
    h0 := h0 + a
    h1 := h1 + b
    h2 := h2 + c
    h3 := h3 + d
    h4 := h4 + e
    h5 := h5 + f
    h6 := h6 + g
    h7 := h7 + h

Produce the final hash value (big-endian):
hash := h0 append h1 append h2 append h3 append h4 append h5 append h6 append h7
digest := hash
```