CS 265 Project Report

SHA1 hashing implementation

&

Survey of why SHA1 is considered broken

Guided By: Dr. Thomas Austin

Submitted by

Nayana D V

Shruti Sharma

Sushma Satyanarayan

Table of Contents

1. Introduction

A cryptographic hash function is a hash function that takes an arbitrary block of data and returns a fixed-size bit string, the cryptographic hash value, such that any (accidental or intentional) change to the data will (with very high probability) change the hash value.

SHA stands for "secure hash algorithm". SHA-1 is a cryptographic hash function designed by the United States National Security Agency and published by NIST. It is the improvement upon the original SHA0 and was first published in 1995.An algorithm that takes input data and irreversibly creates a digest of that data is called a one-way hash function. One of the most studied and trusted one-way hash functions is SHA-1. It produces a 160-bit (20-byte) hash value. A SHA-1 hash value typically forms a hexadecimal number, 40 digits long. SHA-1 is the most widely used of the existing SHA hash functions, and is employed in several widely used applications and protocols.

In recent year's cryptanalysts have found attacks on SHA-1 suggesting that the algorithm might not be secure. Few researchers announced that they have found new ways to exploit weaknesses in the collision-resistance of SHA-1. This affects the computational effort that it takes to find (create) two different sets of data that have the same SHA-1 hash value. But SHA-1 is still considered safe to use for most applications. In this project we are implementing the algorithm for creating hashes using SHA1 and doing a survey of literature explaining why SHA-1 is broken.

2. Why is SHA1 viable?

- SHA1 is similar in design to MD5 since both have derived from MD4. But SHA1 is more secure and provides cryptographic integrity and strong collision resistance.
- Most of the protocols like SSL, TLS, SSH, IPSec use SHA1 and it is also used extensively by Digital Signature Standard (DSS) for digitally signing documents.
- GIT, Mercurial and other revision control systems use SHA1 to detect tampering. WII uses it for signature verification while booting.
- The 160 big digest which is generated is tough to break by Brute force.
- It is not vulnerable to known attacks and works well with Big Endian CPU's.
- It's quite tough to find collisions, though the cryptanalysts have found ways to break SHA1.

3. SHA1 Algorithm

- Initialize some variables: There are five variables that now need to be initialized.

  h0 = 01100111010001010010001100000001

  h1 = 11101111110011011010101110001001

  h2 = 10011000101110101101110011111110

  h3 = 00010000001100100101010001110110

  h4 = 11000011110100101110000111110000

- Pick a String : Example A Test
- Break the String into characters

  A

  T

  e

  s

  t

- Convert Characters into ASCII codes
- Convert numbers into Binary
- Append all the numbers together and append 1 at the end. This is done to create a clear demarcation between the 0's that will be appended to it.
- Append 0's at the end: Add zeroes till the length of the message is congruent to 448 mod 512.
- Append 64 bit representation of the original message length in binary to the message
- Chunk the message: Break the message into 512 bit chunks.
- Break the chunks into words: Break the 512 bit chunk into sixteen 32-bit words.
- Extend the 16 words to 80 words : XOR and Left rotation helps in achieving this
- Initialize some variables A → E equal to variables h0 → h4

- Process all the 80 words through 4 functions
  - Words 0-19 : Function 1 = (B AND C) or (!B AND D)
  - Words 20-39 : Function 2 = B XOR C XOR D
  - Words 40-59 : Function 3 = (B AND C) OR (B AND D) OR (C AND D)
  - Words 60-79 : Function 4 = B XOR C XOR D
  - (A left rotate 5) + F + E + K + (the current word)
- Once the main loop processing is done

  h0 = h0 + A

  h1 = h1 + B

  h2 = h2 + C

  h3 = h3 + D

  h4 = h4 + E

- Each of the values h0 → h4 are converted to hexadecimal and appended together to get the final 20 byte hash value

4. Why is SHA1 considered broken?

Though in practicality SHA1 is considered very secure, there has been a lot of debate in this aspect because of the attacks that have considerable reduced the effort required to break the SHA1 hashes.

- SHA1 is processed sequentially
- Word-Expansion is a phase of the SHA1 transformation
- Its purpose is to generate a bigger volume of data out of the input data
- This is where the weakness is located in SHA1
- Input data is mixed up using the following set of logical instructions:
  W[t] = R((W[t-3] ^ W[t-8] ^ W[t-14] ^ W[t-16]), 1)
  W[0] .. W[15] is filled with the input data

- By iterating t from 16 to 79, 2048 additional bits are generated. Listed below are the iterations for 16-29th words.

```
W[16] = R((W[13] ^ W[ 8] ^ W[ 2] ^ W[ 0]), 1)
W[17] = R((W[14] ^ W[ 9] ^ W[ 3] ^ W[ 1]), 1)
W[18] = R((W[15] ^ W[10] ^ W[ 4] ^ W[ 2]), 1)
W[19] = R((W[16] ^ W[11] ^ W[ 5] ^ W[ 3]), 1)
W[20] = R((W[17] ^ W[12] ^ W[ 6] ^ W[ 4]), 1)
W[21] = R((W[18] ^ W[13] ^ W[ 7] ^ W[ 5]), 1)
W[22] = R((W[19] ^ W[14] ^ W[ 8] ^ W[ 6]), 1)
W[23] = R((W[20] ^ W[15] ^ W[ 9] ^ W[ 7]), 1)
W[24] = R((W[21] ^ W[16] ^ W[10] ^ W[ 8]), 1)
W[25] = R((W[22] ^ W[17] ^ W[11] ^ W[ 9]), 1)
W[26] = R((W[23] ^ W[18] ^ W[12] ^ W[10]), 1)
W[27] = R((W[24] ^ W[19] ^ W[13] ^ W[11]), 1)
W[28] = R((W[25] ^ W[20] ^ W[14] ^ W[12]), 1)
W[29] = R((W[26] ^ W[21] ^ W[15] ^ W[13]), 1)
```

- The password candidate generator needs to hold W[1]..W[15] fixed
- Outside the loop pre-compute W [16]...W[79] ignoring the unknown W[0]
- We call this pre-computed buffer PW[]
- Inside the loop W[0] is changed
    - Since the Word-Expansion process is using XOR, we can apply W[0] to the precomputed buffer at a later stage
    - Using XOR is the root of the problem
    - Logical instructions cannot overflow, but arithmetic ones can
    - If the Word-Expansion had used ADD, it would have been impossible to exploit it
- When iterating W[0] changes is finished, W[1]..W[15] can be changed
- Restart the process with the next pre-computed value of W[16]..W[79]

PW[16]..PW[79] in the outer loop are shown below

```
PW[16] = R(( W[13] ^  W[ 8] ^  W[ 2] ^ W[ 0]), 1)
PW[17] = R(( W[14] ^  W[ 9] ^  W[ 3] ^  W[ 1]), 1)
PW[18] = R(( W[15] ^  W[10] ^  W[ 4] ^  W[ 2]), 1)
PW[19] = R((PW[16] ^  W[11] ^  W[ 5] ^  W[ 3]), 1)
PW[20] = R((PW[17] ^  W[12] ^  W[ 6] ^  W[ 4]), 1)
PW[21] = R((PW[18] ^  W[13] ^  W[ 7] ^  W[ 5]), 1)
PW[22] = R((PW[19] ^  W[14] ^  W[ 8] ^  W[ 6]), 1)
PW[23] = R((PW[20] ^  W[15] ^  W[ 9] ^  W[ 7]), 1)
PW[24] = R((PW[21] ^ PW[16] ^  W[10] ^  W[ 8]), 1)
PW[25] = R((PW[22] ^ PW[17] ^  W[11] ^  W[ 9]), 1)
PW[26] = R((PW[23] ^ PW[18] ^  W[12] ^  W[10]), 1)
PW[27] = R((PW[24] ^ PW[19] ^  W[13] ^  W[11]), 1)
PW[28] = R((PW[25] ^ PW[20] ^  W[14] ^  W[12]), 1)
PW[29] = R((PW[26] ^ PW[21] ^  W[15] ^  W[13]), 1)
```

```
PW[30] = R((PW[27] ^ PW[22] ^ PW[16] ^  W[14]), 1)
PW[31] = R((PW[28] ^ PW[23] ^ PW[17] ^  W[15]), 1)
PW[32] = R((PW[29] ^ PW[24] ^ PW[18] ^ Pw[16]), 1)
PW[33] = R((PW[30] ^ PW[25] ^ PW[19] ^ PW[17]), 1)
PW[34] = R((PW[31] ^ PW[26] ^ PW[20] ^ PW[18]), 1)
PW[35] = R((PW[32] ^ PW[27] ^ PW[21] ^ PW[19]), 1)
PW[36] = R((PW[33] ^ PW[28] ^ PW[22] ^ PW[20]), 1)
PW[37] = R((PW[34] ^ PW[29] ^ PW[23] ^ PW[21]), 1)
PW[38] = R((PW[35] ^ PW[30] ^ PW[24] ^ PW[22]), 1)
PW[39] = R((PW[36] ^ PW[31] ^ PW[25] ^ PW[23]), 1)
PW[40] = R((PW[37] ^ PW[32] ^ PW[26] ^ PW[24]), 1)
PW[41] = R((PW[38] ^ PW[33] ^ PW[27] ^ PW[25]), 1)
…
PW[79] = R((PW[76] ^ PW[71] ^ PW[65] ^ PW[63]), 1)
```

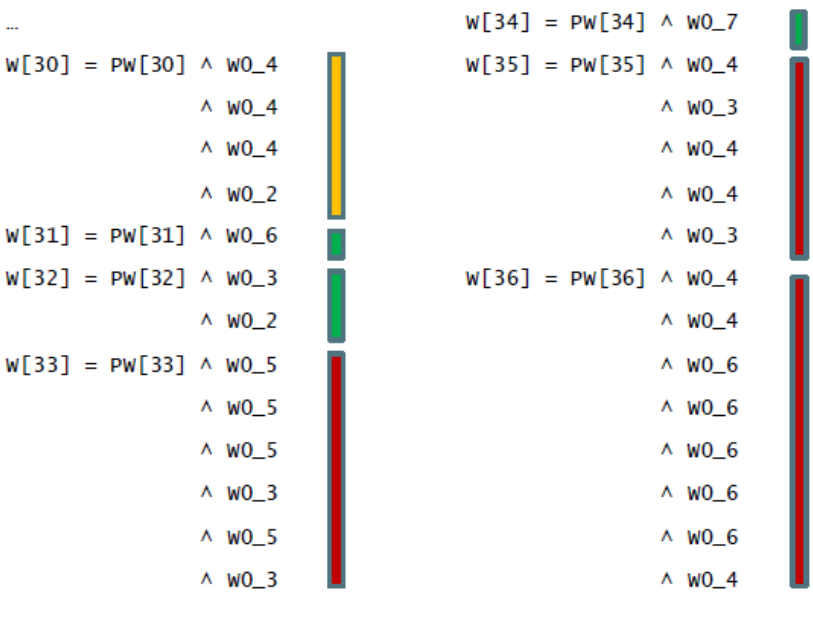
W[0] in the inner loop

```
W0_1 = R(w[0],  1)
W0_2 = R(w[0],  2)
…
W020 = R(w[0], 20)
```

For 1..20 compute R(W[0], i)

```
W[16] = R((w[13] ^ w[ 8] ^ w[ 2] ^ w[ 0]), 1) = PW[16] ^ W0_1
W[17] = R((w[14] ^ w[ 9] ^ w[ 3] ^ w[ 1]), 1) = PW[17]
W[18] = R((w[15] ^ w[10] ^ w[ 4] ^ w[ 2]), 1) = PW[18]
W[19] = R((w[16] ^ w[11] ^ w[ 5] ^ w[ 3]), 1) = PW[19] ^ W0_2
W[20] = R((w[17] ^ w[12] ^ w[ 6] ^ w[ 4]), 1) = PW[20]
W[21] = R((w[18] ^ w[13] ^ w[ 7] ^ w[ 5]), 1) = PW[21]
W[22] = R((w[19] ^ w[14] ^ w[ 8] ^ w[ 6]), 1) = PW[22] ^ W0_3
W[23] = R((w[20] ^ w[15] ^ w[ 9] ^ w[ 7]), 1) = PW[23]
W[24] = R((w[21] ^ w[16] ^ w[10] ^ w[ 8]), 1) = PW[24] ^ W0_2
W[25] = R((w[22] ^ w[17] ^ w[11] ^ w[ 9]), 1) = PW[25] ^ W0_4
W[26] = R((w[23] ^ w[18] ^ w[12] ^ w[10]), 1) = PW[26]
```

Word expansion using Pre-compute



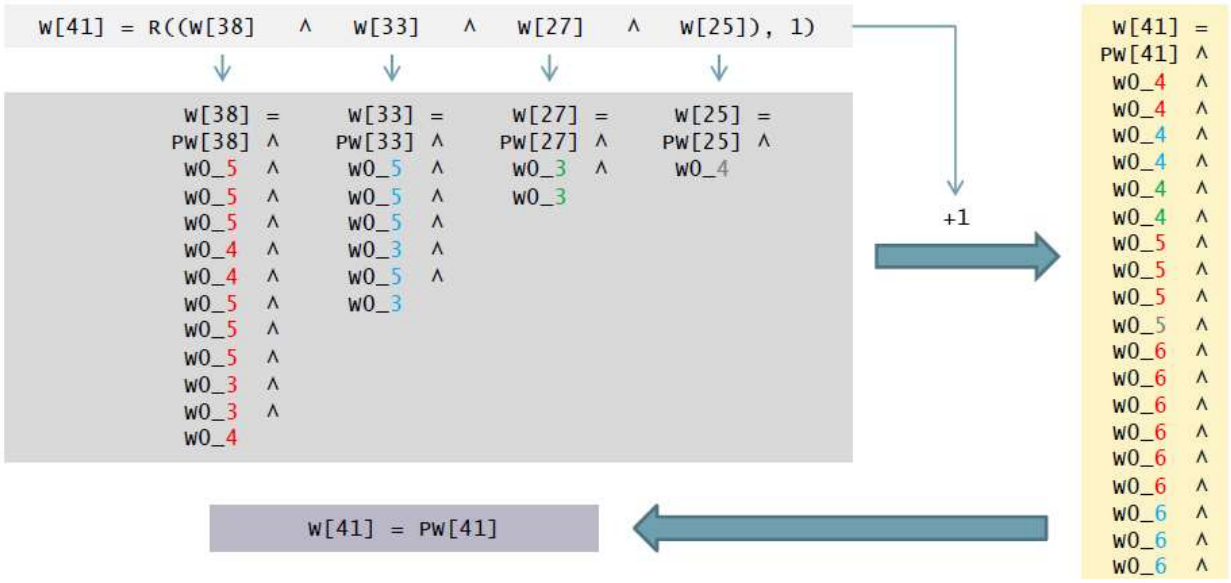
< 4 operations
= 4 operations
> 4 operations

Number of Operations:

W[16] = 1
W[17] = 0
...
W[33] = 6
...
W[43] = 308
...
W[75] = 4703
...

- XORing a value to itself, results in 0
- XORing a value with 0, results in the same value
- We can ignore many XOR operations in order to optimize the procedure

Word expansion / XOR Zeroes



Word Expansion / XOR groups

```
W[36] = PW[36] ^ W0_6 ^ W0_4
W[51] = PW[51] ^ W0_6 ^ W0_4
W[62] = PW[62] ^ W0_6 ^ W0_4 ^ W012 ^ W0_8
…
```

```
const int W0_6___W0_4 = W0_6 ^ W0_4
```

```
…
W[36] = PW[36] ^ W0_6___W0_4
W[51] = PW[51] ^ W0_6___W0_4
W[62] = PW[62] ^ W0_6___W0_4 ^ W012 ^ W0_8
```

Final optimized Word - Expansion

Reference Impl.

```
W[16] = R((W[13] ^ W[ 8] ^ W[ 2] ^ W[ 0]), 1)
W[17] = R((W[14] ^ W[ 9] ^ W[ 3] ^ W[ 1]), 1)
W[18] = R((W[15] ^ W[10] ^ W[ 4] ^ W[ 2]), 1)
W[19] = R((W[16] ^ W[11] ^ W[ 5] ^ W[ 3]), 1)
W[20] = R((W[17] ^ W[12] ^ W[ 6] ^ W[ 4]), 1)
W[21] = R((W[18] ^ W[13] ^ W[ 7] ^ W[ 5]), 1)
W[22] = R((W[19] ^ W[14] ^ W[ 8] ^ W[ 6]), 1)
W[23] = R((W[20] ^ W[15] ^ W[ 9] ^ W[ 7]), 1)
W[24] = R((W[21] ^ W[16] ^ W[10] ^ W[ 8]), 1)
W[25] = R((W[22] ^ W[17] ^ W[11] ^ W[ 9]), 1)
W[26] = R((W[23] ^ W[18] ^ W[12] ^ W[10]), 1)
W[27] = R((W[24] ^ W[19] ^ W[13] ^ W[11]), 1)
W[28] = R((W[25] ^ W[20] ^ W[14] ^ W[12]), 1)
W[29] = R((W[26] ^ W[21] ^ W[15] ^ W[13]), 1)
W[30] = R((W[27] ^ W[22] ^ W[16] ^ W[14]), 1)
```

Optimized Impl.

```
W[16] = PW[16] ^ W0_1
W[17] = PW[17]
W[18] = PW[18]
W[19] = PW[19] ^ W0_2
W[20] = PW[20]
W[21] = PW[21]
W[22] = PW[22] ^ W0_3
W[23] = PW[23]
W[24] = PW[24] ^ W0_2
W[25] = PW[25] ^ W0_4
W[26] = PW[26]
W[27] = PW[27]
W[28] = PW[28] ^ W0_5
W[29] = PW[29]
W[30] = PW[30] ^ W0_4 ^ W0_2
```

Section	Instruction count	Optimization
Unoptimized	880	0 %
- Known optimizations	828	5.1 %
- This weakness, exploited	694	21.1 %

- **Claim1**: A German hacker claims to have successfully cracked a six-character implementation of the 160-bit SHA-1 crypto algorithm using Amazon cloud computing resource. As he states, the hack was completed in 49 minutes at a cost of just $2.10.
  Jon Callas, PGP's CTO, put it best: "It's time to walk, but not run, to the fire exits. You don't see smoke, but the fire alarms have gone off."
- **Claim 2: Eurocrypt 2009 convention:** Exploiting the weakness of SHA1, the practical complexity of breaking SHA1 has been reduced to 2^63. (Latest: 2^33). Attacks can be performed using Computer clusters.

Cracking hash function

For a hash function for which L is the number of bits in the message digest, finding a message that corresponds to a given message digest can always be done using a brute force search in approximately 2^L evaluations. This is called a preimage attack and may or may not be practical depending on L and the

particular computing environment. The second criterion, finding two different messages that produce the same message digest, namely a *collision*, requires on average only about $1.2 * 2^{L/2}$ evaluations using a birthday attack. For the latter reason the strength of a hash function is usually compared to a symmetric cipher of half the message digest length. Thus SHA-1 was originally thought to have 80-bit strength.

Cryptographers have produced collision pairs for SHA-0 and have found algorithms that should produce SHA-1 collisions in far fewer than the originally expected 2^{80} evaluations. In terms of practical security, a major concern about these new attacks is that they might pave the way to more efficient ones.

Dictionary and Brute force attack

The simplest way to crack a hash is to try to guess the password, hashing each guess, and checking if the guess's hash equals the hash being cracked. If the hashes are equal, the guess is the password. The two most common ways of guessing passwords are **dictionary attacks** and **brute-force attacks**.

A dictionary attack uses a file containing words, phrases, common passwords, and other strings that are likely to be used as a password. Each word in the file is hashed, and its hash is compared to the password hash. If they match, that word is the password. These dictionary files are constructed by extracting words from large bodies of text, and even from real databases of passwords. Further processing is often applied to dictionary files, such as replacing words with their "leet speak" equivalents ("hello" becomes "h3110"), to make them more effective.

A brute-force attack tries every possible combination of characters up to a given length. These attacks are very computationally expensive, and are usually the least efficient in terms of hashes cracked per processor time, but they will always eventually find the password. Passwords should be long enough that searching through all possible character strings to find it will take too long to be worthwhile.

There is no way to prevent dictionary attacks or brute force attacks. They can be made less effective, but there isn't a way to prevent them altogether. If your password hashing system is secure, the only way to crack the hashes will be to run a dictionary or brute-force attack on each hash.

Lookup Tables

Lookup tables are an extremely effective method for cracking many hashes of the same type very quickly. The general idea is to **pre-compute** the hashes of the passwords in a password dictionary and store them, and their corresponding password, in a lookup table data structure. A good implementation of a lookup table can process hundreds of hash lookups per second, even when they contain many billions of hashes.

Reverse Lookup Tables

This attack allows an attacker to apply a dictionary or brute-force attack to many hashes at the same time, without having to pre-compute a lookup table.

First, the attacker creates a lookup table that maps each password hash from the compromised user account database to a list of users who had that hash. The attacker then hashes each password guess and uses the lookup table to get a list of users whose password was the attacker's guess. This attack is especially effective because it is common for many users to have the same password.

Rainbow Tables

Rainbow tables are a time-memory trade-off technique. They are like lookup tables, except that they sacrifice hash cracking speed to make the lookup tables smaller. Because they are smaller, the solutions to more hashes can be stored in the same amount of space, making them more effective.

Adding Salt

Lookup tables and rainbow tables only work because each password is hashed the exact same way. If two users have the same password, they'll have the same password hashes. We can prevent these attacks by randomizing each hash, so that when the same password is hashed twice, the hashes are not the same.

The hashes can be randomized by appending or prepending a random string, called a **salt**, to the password before hashing. To check if a password is correct, we need the salt, so it is usually stored in the user account database along with the hash, or as part of the hash string itself.

The salt does not need to be secret. Just by randomizing the hashes, lookup tables, reverse lookup tables, and rainbow tables become ineffective. An attacker won't know in advance what the salt will be, so they can't pre-compute a lookup table or rainbow table. If each user's password is hashed with a different salt, the reverse lookup table attack won't work either.

Collision Attacks

1) “New collision attacks on SHA-1 based on optimal joint local-collision analysis” by Marc Stevens
In this paper, a novel direction in the cryptanalysis of SHA-1 is presented. Secondly, an identical-prefix

collision attack and a chosen-prefix collision attack on SHA-1 with complexities equivalent to approximately 2^61 and 2^77.1 SHA-1 compressions is presented.

A series of breakthrough attacks on hash functions started in 2004 when the first collisions for MD4, MD5, HAVAL-128 and RIPEMD were presented by Wang et al. This was soon followed by the first SHA-0 collision by Biham et al. Soon thereafter, Wang et al. published a more efficient collision attack on SHA-0. In the same year, the first collision attack on full SHA-1 was presented by Wang et al with an estimated complexity of 2^69 compressions. This paper aims to renew the cryptanalytic efforts to construct a feasible collision attack on SHA-1 and find an actual collision pair.

New collision attacks on SHA-1 that are discussed in the paper are:

- Open-source near collision attack
- Identical-prefix collision attack on SHA-1
- Chosen-prefix collision attack

2) A researcher has devised a method that reduces the time and resources required to crack passwords that are protected by the SHA1 cryptographic algorithm. The optimization, presented on Tuesday at the Passwords^12 conference in Oslo, Norway, can speed up password cracking by 21 percent. The optimization works by reducing the number of steps required to calculate SHA1 hashes, which are used to cryptographically represent strings of text so passwords aren't stored as plain text. Such one-way hashes—for example 5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8 to represent "password and e38ad214943daad1d64c102faec29de4afe9da3d for "password1"—can't be mathematically unscrambled, so the only way to reverse one is to run plaintext guesses through the same cryptographic function until an identical hash is generated.

Details of attacks on SHA1

- **2011** [RFC6194]: first attack is best attack
- **2012** New results in [thesis]
  - Exact joint local-collision analysis
  - Preliminary near-collision attack: 257.5 calls
  - Extends to identical- & chosen-prefix collision

5. References

[1] http://m.metamorphosite.com/one-way-hash-encryption-sha1-data-software

[2] http://en.wikipedia.org/wiki/SHA-1

[3] http://www.sha1-online.com/

[4] http://stackoverflow.com/questions/8929357/encryption-using-sha1

[5] http://m.metamorphosite.com/one-way-hash-encryption-sha1-data-software

[6] http://2oj.com/2010/02/generating-sha-1-hash-using-java/

[7] http://www.mkyong.com/java/java-sha-hashing-example/

[8] https://www.schneier.com/blog/archives/2005/02/sha1_broken.html

[9] http://www.infosecurity-magazine.com/view/14059/sha1-crypto-protocol-cracked-using-amazon-cloud-computing-resources/

[10] http://crypto.stackexchange.com/questions/3690/no-sha-1-collision-yet-sha1-is-broken

[11] http://stackoverflow.com/questions/2772014/is-sha-1-secure-for-password-storage

[12] https://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html

[13] http://arstechnica.com/security/2012/12/oh-great-new-attack-makes-some-password-cracking-faster-easier-than-ever/

[14] https://hashcat.net/events/p12/

[15] http://arstechnica.com/security/2012/06/8-million-leaked-passwords-connected-to-linkedin/

[16] https://tools.ietf.org/html/rfc3174

[17] http://johans.livejournal.com/3834.html

[18] http://www.theregister.co.uk/2005/02/17/sha1_hashing_broken/

[19] http://www.zdnet.com/blog/ou/putting-the-cracking-of-sha-1-in-perspective/409

[20] http://tinsology.net/2010/12/is-sha1-still-viable/

[21] http://marc-stevens.nl/research/papers/EC13-S.pdf

[22]http://arstechnica.com/security/2012/12/oh-great-new-attack-makes-some-password-cracking-faster-easier-than-ever/

[23]http://www.cosic.esat.kuleuven.be/publications/article-989.pdf

[24]http://www.codeproject.com/Articles/704865/Salted-Password-Hashing-Doing-it-Right