

SHA-1:

Introduction:

Secure Hashing Algorithm is what SHA stands for and the 1 stands for the first version. Now what is hashing and what is its significance. Hashing refers to the process of taking an input string and applying a mathematical function to transform it into a fixed-length, giving a sequence of characters called a "hash." Giving the same input will always produce the same hash. Hashing is widely used for efficiently storing and indexing data, as it enables quick lookups in large datasets.

A helpful analogy is looking up a word in a dictionary: instead of flipping through all the pages, you rely on the organized structure (like hashing) to quickly find the word.

From a security perspective, hashing becomes crucial when combined with cryptographic algorithms. Cryptographic hashing applies complex mathematical formulas to create hashes that are computationally impossible to reverse, ensuring the original input cannot be easily determined. An example of how cryptographic hashing is used is for password storage, data verification, and digital signatures due to its irreversible and secure nature.

Hashing is a highly effective method of securing data by converting it into a fixed-length, unique string (hash) that cannot be reversed to its original form. It ensures data integrity, as even minor changes to the input produce a completely different hash. Hashing is widely used for password storage, data verification, and digital signatures due to its irreversible and secure nature.

For example SHA-256 will always return a 64 character hash (or string). Hashing allows data to be secure by making it difficult to revert to the original string. In the real world we can commonly see this used as a storage method of passwords. Where only the hash of the password is being stored instead of the raw text of the password. An example on how this works with authentication is that environments will compare hashes instead of the raw text of the password.

History of SHA-1:

SHA-1 was developed by the US government back in 1993 and was a project that was developed by US government standard agencies NIST. Later the 1993 version was superseded and it's now called SHA-0 and was replaced in 1995 with a small change. The only change was a single

bitwise rotation in the message schedule. The reason for this was that the original version contained a flaw which affected “cryptographic security”. [8]

SHA-1 was officially removed from major products as the method of hashing data was now deemed insecure. Its successor (SHA-256) is still commonly used, the main difference being that the hash sizes are larger. The biggest worry with smaller hash sizes is collisions which is when two different inputs create the same hash.

SHA-1 was widely used for many applications and protocols before it was superseded. It was used widely for TLS, SSL, PGP, SSH, S/MIME, and IPsec. This method of hashing was required by law for certain US government applications. This hashing algorithm played a key role in the security infrastructure and still is with its successors.

Intuition:

Hashing Functions:

I decided to conduct research on two different hashing algorithms, one allowing us to go over the concept in a little bit more simpler perspective. And the other being more sophisticated giving more of a real world example of how hashing really works.

DJB2:

The DJB2 hash function is a great example of hashing, where raw text is converted into a numeric representation that cannot be reversed. It initializes a hash value to 5381 and processes each character by shifting the hash 5 bits to the left (effectively multiplying by 32), adding the original hash (to make it a multiplication by 33), and then adding the ASCII value of the character. This simple formula efficiently produces a unique (but not cryptographic) representation of the input text.[6]

Bitwise Operations:

Before we begin explaining the steps of how SHA-1 works I think it would be useful to go over some bitwise operations. SHA-1 uses bitwise AND, OR, XOR, NOT, and circular shift operations. Which are used to mix bits, maintain fixed block sizes, and utilize non-linear transformations. [9]

AND : Returns true if both conditions are true.

OR : Returns true if atleast one condition is true.

NOT: Negates the value. True becomes false vice versa.

XOR : Returns true if exactly one condition is true.

Left shift : Shifts bits to the left by filling the vacant bits with 0's.

AND operation	OR operation	NOT operation	XOR operation	Left shift
$1 \& 1 = 1$	$1 1 = 1$	$\sim 1 = 0$	$1 \wedge 1 = 0$	$1111 \ll 2$
$1 \& 0 = 0$	$1 0 = 1$	$\sim 0 = 1$	$1 \wedge 0 = 1$	$= 111100$
$0 \& 0 = 0$	$0 0 = 0$		$0 \wedge 0 = 0$	

SHA-1:

Sha-1 first works by taking in a string input then begins to pad the message. Padding the message is “to ensure that length is congruent to 448 modulo 512”[5]. What is padding? Padding is the processing step where the input message is suitable for an algorithm to process the input.

Allowing the algorithm to divide the message into 512 bit blocks. Then to “divide into 16 words of 32 bits” [5]. Then these words will be “expanded into 80 32-bit words”[5]. After they are split into 80 words then SHA-1 uses five constant values which are applied to a specific range of rounds (0-79). After all rounds the values are all added into their original hash values in order to produce the final hash.

Steps of How SHA-1 Works:

1. First the message needs padding the message by adding 0's until it's a multiple of 512 (or 64 bytes). Then append the original length of the message at the end.
2. Then we will divide the padded message into 16 32 bit “words”
3. Extend the 16 32 bit words to 80 32 bit words. First we are just initializing the 16 words from the message block. By extending these words we will run XOR operations on words before the word we are iterating on and then rotate the bits left by 1 bit.
$$W[i] = \text{RotateLeft}(W[i - 3] \text{ XOR } W[i - 8] \text{ XOR } W[i - 14] \text{ XOR } W[i - 16], 1)$$
4. We set up the five working variables with the given five constant values; $A=H_0$, $B=H_1$, $C=H_2$, $D=H_3$, and $E=H_4$. We are cycling through 80 rounds applying different operations to each word. For this we have a round function ‘F’ which determines how intermediate hash values B,C,D are combined each round. As well we have a different constant value of ‘K’ which is applied differently to each group of 20 rounds.

Values of K ;

Rounds 0-19: $K=0x5A827999$

Rounds 20-39: $K=0x6ED9EBA1$

Rounds 40-59: $K=0x8F1BBCDC$

Rounds 60-79: $K=0xCA62C1D6$

Function of F:

Rounds 0-19 : $F = (B \& C) \mid ((\sim B) \& D)$

Rounds 20-39: $F = B \wedge C \wedge D$

Rounds 40-59: $F = (B \& C) \mid (B \& D) \mid (C \& D)$

Rounds 60-79: $F = B \wedge C \wedge D$

Key: $\& = \text{and}$, $\mid = \text{or}$, $\sim = \text{not}$, $\wedge = \text{XOR}$

For each round we store these values into a temp value. F adds the logic and complexity and K introduces the round specific constants. By combining this with words (word[iteration]) this ensures that hash computation is resistant to prediction.

$TEMP = \text{RotateLeft}(A, 5) + F + E + W[\text{iteration}] + K$

$E = D$, $D = C$, $C = \text{RotateLeft}(B, 30)$, $B = A$, $A = TEMP$

5. After 80 rounds we get the final values for A,B,C,D,E. We now update our constant the five constant values of H by adding the final values. $H0 += A$, $H1 += B$, $H2 += C$, $H3 += D$, and $H4 += E$. After we add to all the blocks H0-H4 contain all the final hash values. To create the final hash we concatenate them into one 160-bit single value. Concatenate means joining each binary value end to end.

Pseudocode:

DJB2:

```
FUNCTION djb2Hash(inputString):  
    hash = 5381 // Initial seed value  
    FOR each character c IN inputString:  
        hash = (hash << 5) + hash + c // Equivalent to hash * 33 + c  
    RETURN hash
```

Argument for Correctness

1. Efficient Hashing:

- The algorithm uses a fixed multiplier (33) that has been found to provide good

distribution for hash values with minimal collisions.

Example of execution:

[illegible]

SHA-1:

padded_message:

```
func padded_message(message):
    padded_message = [] // Initialize an empty list for the padded message

    // Append the original message to the padded message
    padded_message += message

    // Append a single '1' bit (0x80 in hexadecimal is 10000000 in binary)
    padded_message += 0x80

    // Add zero bytes (0x00) until the message length is 64 bits shy of a multiple of 512
    while ((size(padded_message) * 8) % 512 != 448):
        padded_message += 0x00

    // Append the original message length as a 64-bit big-endian integer
    original_length = size(message) * 8 // Calculate the length of the original message in
bits
    for i from 7 to 0: // Iterate from 7 to 0 for the 8 bytes of the 64-bit integer
        padded_message += (original_length >> (i * 8)) & 0xFF // Extract each byte

    return padded_message
```

Example of execution:

[illegible]

parse_message:

```
func parse_message(padded_message):  
    blocks = [] // Initialize an empty list to store 32-bit words  
  
    // Iterate over the padded_message in chunks of 4 bytes (32 bits)  
    for i from 0 to size(padded_message) - 1 step 4:  
        block = 0 // Initialize the block as 0  
  
        // Combine 4 consecutive bytes into a single 32-bit word  
        for j from 0 to 3:  
            block = (block << 8) | padded_message[i + j]  
  
        blocks += block // Add the 32-bit word to the list of blocks  
  
    return blocks
```

Example of Execution:

[illegible]

leftRotate:

```
func leftRotate(value , shift):  
    return (value << shift) | (value >> (32 - shift))
```

sha1:

```
func sha1(message):  
    // Step 1: Preprocessing  
    padded_message = pad_message(message)  
    blocks = parse_message(padded_message)  
  
    // Step 2: Initialize hash values (H0, H1, H2, H3, H4)  
    H0 = 0x67452301  
    H1 = 0xEFCDAB89  
    H2 = 0x98BADCFE  
    H3 = 0x10325476  
    H4 = 0xC3D2E1F0  
  
    // Step 3: Process each 512-bit block  
    for block in blocks:  
        // Step 3.1: Prepare the message schedule (W)  
        W = array[80]  
        for i from 0 to 15:  
            W[i] = block[i] // First 16 words come from the block  
        for i from 16 to 79:  
            W[i] = leftRotate(W[i-3] XOR W[i-8] XOR W[i-14] XOR W[i-16], 1)  
  
        // Step 3.2: Initialize working variables (a, b, c, d, e)  
        a = H0  
        b = H1  
        c = H2  
        d = H3  
        e = H4  
  
        // Step 3.3: Perform the main compression loop  
        for i from 0 to 79:  
            if i between 0 and 19:  
                f = (b AND c) OR ((NOT b) AND d)  
                k = 0x5A827999  
            else if i between 20 and 39:  
                f = b XOR c XOR d  
                k = 0x6ED9EBA1  
            else if i between 40 and 59:
```

```

        f = (b AND c) OR (b AND d) OR (c AND d)
        k = 0x8F1BBCDC
    else if i between 60 and 79:
        f = b XOR c XOR d
        k = 0xCA62C1D6

    temp = (leftRotate(a, 5) + f + e + k + W[i]) & 0xFFFFFFFF
    e = d
    d = c
    c = leftRotate(b, 30)
    b = a
    a = temp

// Step 3.4: Add the working variables back into the hash values
H0 = (H0 + a) & 0xFFFFFFFF
H1 = (H1 + b) & 0xFFFFFFFF
H2 = (H2 + c) & 0xFFFFFFFF
H3 = (H3 + d) & 0xFFFFFFFF
H4 = (H4 + e) & 0xFFFFFFFF

// Step 4: Produce the final hash value
return concatenate(H0, H1, H2, H3, H4) as hexadecimal string

```

Argument for Correctness

1. Adheres to the SHA-1 Standard:

- The algorithm follows the structure and steps defined by the SHA-1 cryptographic hash standard, including padding, message scheduling, and the main processing loop.

2. Handles Any Input:

- The preprocessing step ensures the input is correctly padded, making it compatible with the SHA-1 block size requirements.

3. Secure Message Scheduling:

- The schedule ensures that each word in the block contributes to the final hash, even for long inputs, by using XOR and left rotation.

4. Iterative Hashing:

- Each block updates the hash values incrementally, preserving all information

from previous blocks.

5. Comparing Hashes:

- If we run SHA-1 algorithm on two different occasions with the same input string. We should in theory get the same output hash as we are performing the
- same operations on both strings.

Example of execution of 80 rounds of operations performed on values A,B,C,D E:

[illegible]

Example of execution after all 80 rounds and we are concatenating the final values of

```
after 80 rounds we get
A :01101101100011001111000100111011 B :11101110011101001100001100100101
C :01000011010110000011001000000011 D :0000100011000110001101111101001
E :0101110100111110101110110111110
SHA-1 Hash: 30cbeb3351ffa43a3460a35336af11ca49181053
```

Asymptotic Runtime Analysis:

DJB2:

Runtime: **$O(n)$**

This is because we only have one loop that is iterating through the size of the string.

SHA-1:

Runtime: **$O(n)$**

First step of this algorithm is pre pre-processing; The *padMessage* function has a while loop that iterates until the size input string. Then *parseMessage* iterates to the given size of the padded message. The main function *SHA-1* will iterate to the size of *parseMessage* by 512 bit blocks.

Analysis:

1. **Block Count:** For the main loop we iterate through the block size of 512 bits. So $O(n/512) = O(n)$.
 - a. **Preprocessing:** Padding and parsing through the message would have a $O(n)$ runtime due to n being the size of the input message.
2. **Per-Block processing:** Each block requires 80 rounds of operations. So the cost per block is $O(80) = O(1)$.

Total Cost: $O(n) * O(1) = O(n)$

Sources:

1. <https://tresorit.com/blog/the-history-of-encryption-the-roots-of-modern-day-cyber-security/#:~:text=The%20first%20encryption%20was%20the,with%20the%20sender%20and%20receiver.>
2. <https://spectrum.ieee.org/hans-peter-luhn-and-the-birth-of-the-hashing-algorithm>

3. <https://www.clickssl.net/blog/difference-between-hashing-vs-encryption#:~:text=Hashing%20vs%20Encryption%20%E2%80%93%20Hashing%20refers,the%20reach%20of%20thir%20parties.>
4. <https://en.wikipedia.org/wiki/SHA-1>
5. <https://www.geeksforgeeks.org/sha-1-hash-in-java/>
6. <http://www.cse.yorku.ca/~oz/hash.html>
7. <https://chatgpt.com/>
8. [https://en.wikipedia.org/wiki/SHA-1#:~:text=SHA%2D1%20was%20developed%20as,Institute%20of%20Standards%20and%20Technology\).](https://en.wikipedia.org/wiki/SHA-1#:~:text=SHA%2D1%20was%20developed%20as,Institute%20of%20Standards%20and%20Technology).)
9. <https://www.geeksforgeeks.org/bitwise-operators-in-c-cpp/>