**Module on Cryptographic Hash Functions**

Kimberly Lluz

1. **Abstract**

Hash algorithms are foundation to many cryptographic applications. The hash algorithm has a variety of other uses even though it is widely linked with digital signature technology. SHA-1 is amongst the most widely known, trusted and used. This module aims to enhance understanding on Hash functions by creating a hashing application using SHA1. The specifications of this application were chosen to allow us to spend more time on computational thinking while figuring out and communicating well-annotated, structured, and mathematically correct solutions to quantitative problems.

1. **Introduction**

Hash function is one of the most commonly used encryption methods. A hash is a special mathematical function that performs one-way encryption. It is a function that takes a relatively arbitrary amount of input and produces an output of fixed size. The properties of some hash functions can be used to greatly increase the security of a system administrator’s network; when implemented correctly they can verify the integrity and source of a file, network packet, or any arbitrary data. The standard hash function serves as a basis for the discussion of Cryptographic Hash Functions.

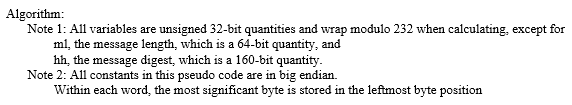
Security Hash Algorithm (SHA) was designed as the algorithm to be used for secure hashing in the US Digital Signature Standard. This module aims to implement the SHA-1 cryptographic hash algorithm using Java. Although SHA-1 has been deemed insecure due to recently verified forced collision attacks, it remains to be an interesting algorithm.

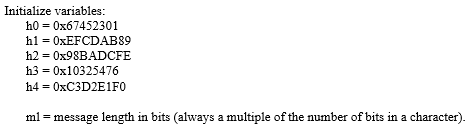
1. **Backgrounder**

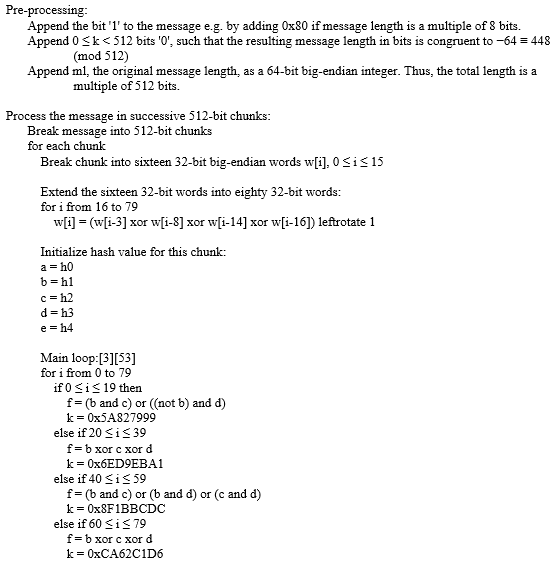
A cryptographic hash function is a kind of algorithm that can be run on a piece of data, like an individual file or a password, producing a value called a checksum. The main use of a cryptographic hash function is to verify the authenticity of a piece of data. Two files can be assured to be identical only if the checksums generated from each file, using the same cryptographic hash function, are identical. Some commonly used cryptographic hash functions include [MD5](https://www.lifewire.com/what-is-md5-2625937) and [SHA-1](https://www.lifewire.com/what-is-sha-1-2626011), though many others also exist.

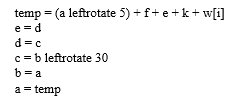
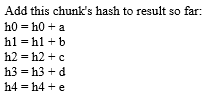
SHA1 stands for “Secure Hash Algorithm 1”, it is the first revision of a hash algorithm developed by the National Security Agency. SHA1 supports messages of any length less than 2 64 bits as input, and produces a 160-bit digest. In the unlikely event that one wishes to compute the digest of a message larger than 264 bits in length (over 2 billion GB of information), the simplest solution would be to divide the large messages into smaller messages. There are no known weaknesses in SHA1, and it is generally considered the more secure of the two algorithms. There are also variations of SHA1 which produce longer digests, SHA-256, SHA-512. They produce digests of 256 bits and 512 bits, respectively.

The SHA-1 is called secure because it is computationally infeasible to find a message which corresponds to a given message digest, or to find two different messages which produce the same message digest. Any change to a message in transit will, with very high probability, result in a different message digest, and the signature will fail to verify.





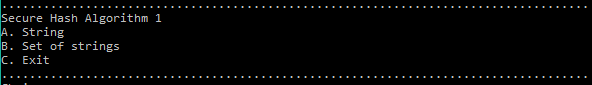




1. **Report proper**

Sample Run:



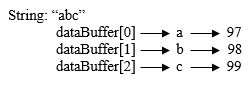
First thing to do is to initialize arrays H and K.



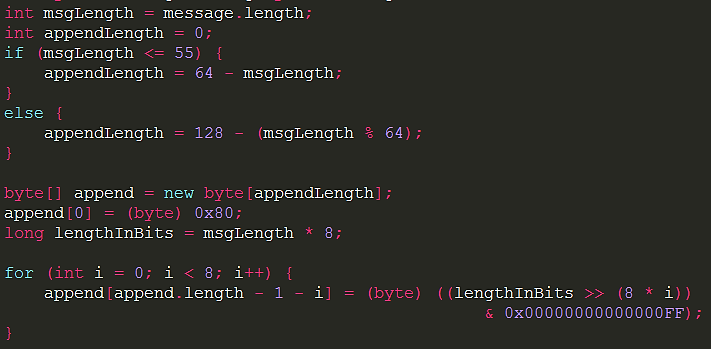
If the first option String is chosen, the user is asked to input a string. The message in the inputted string will then be encoded into a sequence of bytes and returns an array of bytes with the getBytes() method. Each character of the string is converted into its ASCII value of byte.

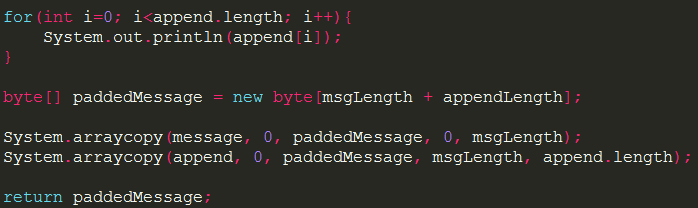


Example:

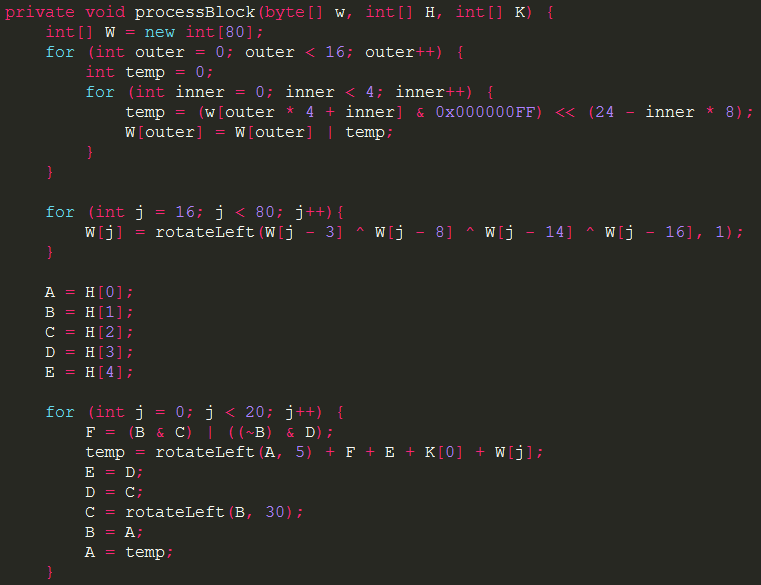
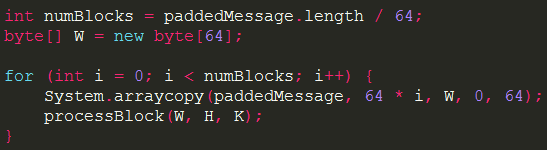


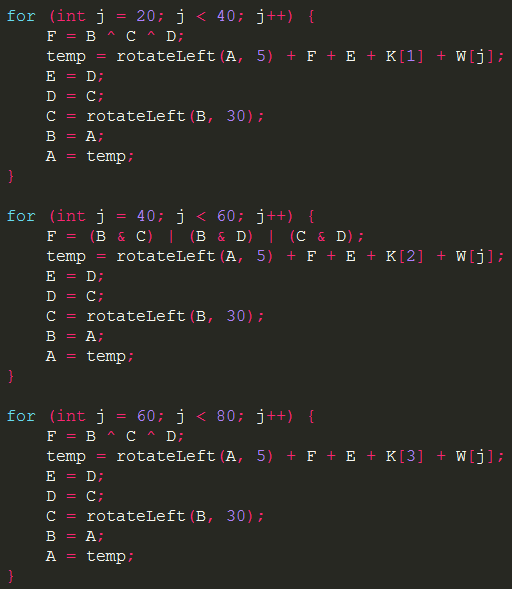
The byte array of the string is then padded in the messagePadding() method. Message padding is done by appending a 1 to the string of characters that are converted to their hexadecimal values, followed by enough 0s until message is 448 bits. The length of the message represented by 64 bits is then added to the end, producing a message that is 512 bits long. The first step in the method is to identify how many more bytes must be appended. The message length must be 55 or less (55 bytes is equal to 440 bits plus 8 bits (append 1) and the last 64 bits for the message length, having a total of 512) to have a one 512-bit block. It must be subtracted from 64 bytes (512 bit) to know the length that will be appended. However, for messages with length more than 55, the length to append is found by 128 subtracted to the message length modulo 64. Message length modulo 64 is the remaining bytes left from the 512 bit blocks that are formed from the message. 128(two 512-bit/64-byte blocks) because the first block contains the message and the second block will contain the remaining bytes of the message and the appended message. Appending of the message will then take place. Below is a code snippet of the messagePadding() method.

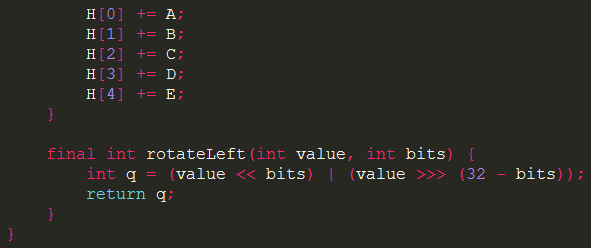




After the message is padded, the padded message will then be divided int 64-byte blocks through the code snippet below.







After all the blocks are processed, the final H array will be converted to string and it will be the final output that will printed.

If the set of strings is the chosen option, the split method is used to split the inputted strings into a string array. Per string of the array are padded and processed, and the results are then joined together.

1. **Result and Discussion**

This section presents the output of the SHA-1 algorithm on two different inputs.

A. Example 1

This example consists of the input string “abc”. The hexadecimal equivalent of this string is “01100001 01100010 01100011”. The length of this string is 24. We first append a “1” to the hexadecimal representation of “abc”. We then append the appropriate number of 0’s followed by the 64-bit binary representation of the length of the string. In this case we have only 1 block of length 512 bits.

The different words of this block are:

W[0] = 61626380

W[1] = 00000000

W[2] = 00000000

W[3] = 00000000

W[4] = 00000000

W[5] = 00000000

W[6] = 00000000

W[7] = 00000000

W[8] = 00000000

W[9] = 00000000

W[10] = 00000000

W[11] = 00000000

W[12] = 00000000

W[13] = 00000000

W[14] = 00000000

W[15] = 00000018

After processing the block, we get the values of Hi as,

H0 = 67452301 + 42541b35 = a9993e36

H1 = efcdab89 + 5738d5e1 = 4706816a

H2 = 98badcfe + 21834873 = ba3e2571

H3 = 10325476 + 681e6df6 = 7850c26c

H4 = c3d2e1f0 + d8fdf6ad = 9cd0d89d

The digest is: a9993e36 4706816a ba3e2571 7850c26c 9cd0d89d

B. Example 2

This example consists of the string “abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq”. The length of this string is 448. We first append a “1” to the hexadecimal representation of “abc”. We then append the appropriate number of 0’s followed by the 64-bit binary representation of the length of the string. In this case we have only 2 blocks of length 512 bits.

The different words of block 1 are:

W[0] = 61626364

W[1] = 62636465

W[2] = 63646566

W[3] = 64656667

W[4] = 65666768

W[5] = 66676869

W[6] = 6768696a

W[7] = 68696a6b

W[8] = 696a6b6c

W[9] = 6a6b6c6d

W[10] = 6b6c6d6e

W[11] = 6c6d6e6f

W[12] = 6d6e6f70

W[13] = 6e6f7071

W[14] = 80000000

W[15] = 00000000

After processing block 1, we get the values of Hi as,

H0 = 67452301 + 8ce34517 = f4286818

H1 = efcdab89 + d3ad7c25 = c37b27ae

H2 = 98badcfe + 6b4e1883 = 0408f581

H3 = 10325476 + 74351cd2 = 84677148

H4 = c3d2e1f0 + 86838382 = 4a566572

The words of block 2 are:

W[0] = 00000000

W[1] = 00000000

W[2] = 00000000

W[3] = 00000000

W[4] = 00000000

W[5] = 00000000

W[6] = 00000000

W[7] = 00000000

W[8] = 00000000

W[9] = 00000000

W[10] = 00000000

W[11] = 00000000

W[12] = 00000000

W[13] = 00000000

W[14] = 00000000

W[15] = 000001C0

After processing block 2, the values of Hi are,

H0 = 67452301 + 906fd62c = 84983e44

H1 = efcdab89 + 58c0aac0 = 1c3bd26e

H2 = 98badcfe + b6a55520 = baae4aa1

H3 = 10325476 + 74e9b89d = f95129e5

H4 = c3d2e1f0 + 9af00b7f = e54670f1

The digest is:

84983e44 1c3bd26e baae4aa1 f95129e5 e54670f1

C. Example 3

This example deals with a string “The quick brown fox jumps over the lazy dog”.

The digest is:

2fd4e1c6 7a2d28fc ed849ee1 bb76e739 1b93eb12

D. Example 4

This example deals with a string “The quick brown fox jumps over the lazy cog”.

The digest is:

de9f2c7f d25e1b3a fad3e85a 0bd17d9b 100db4b3

E. Example 5

This example deals with an empty string “”.

The digest is:

da39a3ee 5e6b4b0d 3255bfef 95601890 afd80709

The running speed of the implemented algorithm was found to be acceptable for the above mentioned examples. This was mainly because of the small size of the input messages. However, it was observed that as the input size went on increasing the program became slower.

For recording purposes, we learnt that for an input message of 10000 characters, it consumed around half a minute to generate the final output. But as we go on higher, the program becomes unacceptably slow. For example, for an input size of around 50000 characters, it generated the message digest in approximately 20 minutes.

1. **Conclusion**

The Secure Hash Algorithm (SHA-1) is used for computing a compressed representation of a message or a data file. Given an input message of arbitrary length < 264 bits, it produces a 160-bit output called the message digest. The SHA-1 algorithm is claimed to be secure because it is practically infeasible to compute the message corresponding to a given message digest. Also it is extremely improbable to detect two messages hashing to the same value.

The basic SHA-1 algorithm was studied with detailed explanation of the alphabet structure used along with various different operators, functions and constants employed by the algorithm.

1. **References**

[1] Bakar, R. (2013, October).*SHA-1 implementation*. Retrieved from http://www.royabubakar.com/blog/2013/10/05/

[2] Brilliant. Secure Hashing Algorithms. Retrieved from https://brilliant.org/wiki/secure-hashing-algorithms/

[3] Eastlake, D. (2001, September). *US Secure Hash Algorithm 1 (SHA1)*. Retrieved from https://tools.ietf.org/html/rfc3174

[4] [Fisher](https://www.lifewire.com/tim-fisher-2617714), T. (2015, October). *Cryptographic Hash Function*. Retrieved from <https://www.lifewire.com/cryptographic-hash-function-2625832>

[5] Jain, R. (2011) .*Cryptography Hash functions*. Retrieved from <http://www.cse.wustl.edu/~jain/cse571-11/>

[6] [Northcutt](http://www.sans.org/instructors/stephen-northcutt), S. *Hash Functions*. Retrieved from <http://www.sans.edu/cyber-research/security-laboratory/article/hash-functions>

[7] Silva, J. (2003, January). *An Overview of Cryptographic Hash Functions and Their Uses*. Retrieved from <https://www.sans.org/reading-room/whitepapers/vpns/overview-cryptographic-hash-functions-879>

[8] Thakur, D. [*SHA-1 - What is Secure Hash Algorithm-l (SHA-1)?*](http://ecomputernotes.com/computernetworkingnotes/security/sha-1). Retrieved from http://ecomputernotes.com/computernetworkingnotes/security/sha-1

[9] Veness, C. (2002). *SHA-1 Cryptographic Hash Algorithm*. Retrieved from http://www.movable-type.co.uk/scripts/sha256.html