Cryptography and Network Security Lab

Assignment 12
Implementation and Understanding of SHA-512 Algorithm

2019BTECS00058
Devang K

Batch: B2

Title: Implementation and Understanding of SHA-512 Algorithm

Aim: To Study, Implement and Demonstrate the SHA-512 Algorithm

Theory:

SHA-2 (Secure Hash Algorithm 2) is a set of cryptographic hash functions designed by the United States National Security Agency (NSA) and first published in 2001. They are built using the Merkle–Damgård construction, from a one-way compression function itself built using the Davies–Meyer structure from a specialized block cipher.

SHA-2 includes significant changes from its predecessor, SHA-1. The SHA-2 family consists of six hash functions with digests (hash values) that are 224, 256, 384 or 512 bits: SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256. SHA-256 and SHA-512 are novel hash functions computed with eight 32-bit and 64-bit words, respectively. They use different shift amounts and additive constants, but their structures are otherwise virtually identical, differing only in the number of rounds. SHA-224 and SHA-384 are truncated versions of SHA-256 and SHA-512 respectively, computed with different initial values. SHA-512/224 and SHA-512/256 are also truncated versions of SHA-512, but the initial values are generated using the method described in Federal Information Processing Standards (FIPS) PUB 180-4.

SHA-2 was first published by the National Institute of Standards and Technology (NIST) as a U.S. federal standard (FIPS). The SHA-2 family of

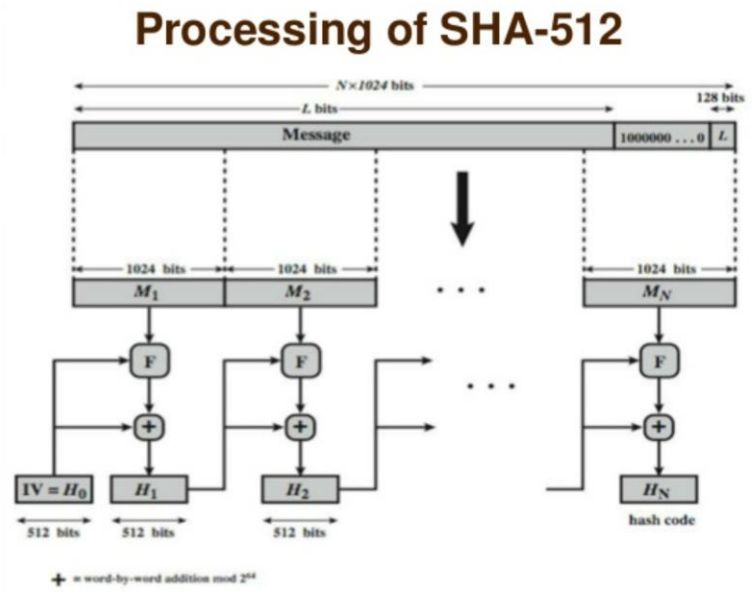
algorithms are patented in US. The United States has released the patent under a royalty-free license.

SHA-512 is a function of cryptographic algorithm SHA-2, which is an evolution of famous SHA-1.

SHA-512 is very close to Sha-256 except that it used 1024 bits "blocks", and accept as input a 2^128 bits maximum length string. SHA-512 also has others algorithmic modifications in comparison with Sha-256.

Pseudocode is as follows:

- the message is broken into 1024-bit chunks
- the initial hash values and round constants are extended to 64 bits
- there are 80 rounds instead of 64
- the message schedule array has 80 – 64-bit words instead of 64 – 32-bit words
- to extend the message schedule array w, the loop is from 16 to 79 instead of from 16 to 63
- the round constants are based on the first 80 primes 2, ..., 409
- the word size used for calculations is 64 bits long
- the appended length of the message (before pre-processing), in bits, is a 128-bit big-endian integer
- the shift and rotate amounts used are different

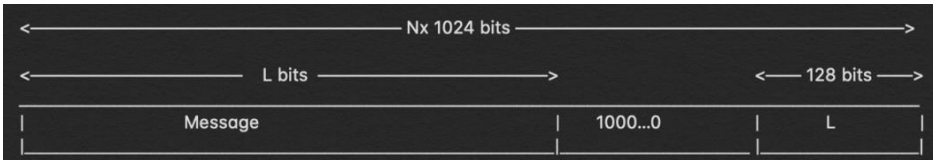


Following is the step-wise working of SHA-512:

1. Appending Bits

The first step is to carry out the padding function in which we append a certain number of bits to the plaintext message to increase its length, which should be exactly 128 bits less than an exact multiple of 1024.

When we are appending these bits at the end of the message we start with a '1' and then keep on adding '0' till the moment we reach the last bit that needs to be appended under padding and leave the 128 bits after that.



2. Append: Length bits

Now, we add the remaining 128 bits left to this entire block to make it an exact multiple of 1024, so that the whole thing can be broken down in 'n' number of 1024 blocks of message that we will apply our operation on. The way to calculate the rest of the 128 bits is by calculating the modulo with 2^{64}.

Once done, we append it to the padded bit and the original message to make the entire length of the block to be "n x 1024" in length.

3. Initialize the buffers

Now, that we have "n x 1024" length bit message that we need to hash, let us focus on the parts of the hashing function itself. To carry on the hash and the computations required, we need to have some default values initialized.

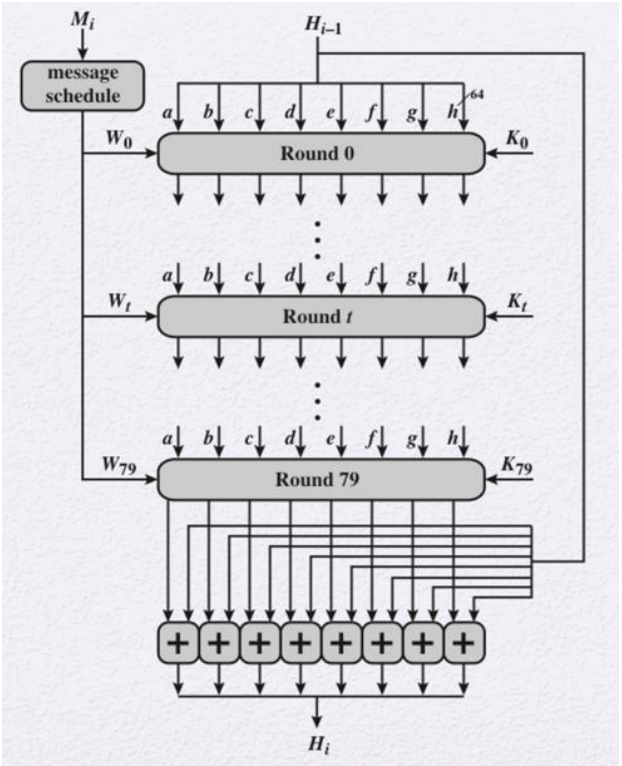
```
a = 0x6a09e667f3bcc908
b = 0xbb67ae8584caa73b
c = 0x3c6ef372fe94f82b
d = 0xa54ff53a5f1d36f1
e = 0x510e527fade682d1
f = 0x9b05688c2b3e6c1f
g = 0x1f83d9abfb41bd6b
h = 0x5be0cd19137e2179
```

These are the values of the buffer that we will need, there are other default values that we need to initialize as well. These are the value for the ‘k’ variable which we will be using.

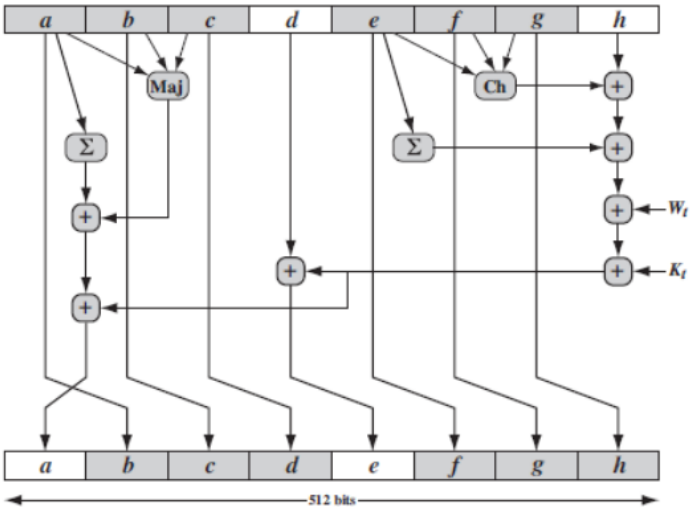
```
k[0..79] := [ 0x428a2f98d728ae22, 0x7137449123ef65cd, 0xb5c0fbcfec4d3b2f, 0xe9b5dba58189dbbc, 0x3956c25bf348b538,
              0x59f111f1b605d019, 0x923f82a4af194f9b, 0xab1c5ed5da6d8118, 0xd807aa98a3030242, 0x12835b0145706fbe,
              0x243185be4ee4b28c, 0x550c7dc3d5ffb4e2, 0x72be5d74f27b896f, 0x80deb1fe3b1696b1, 0x9bdc06a725c71235,
              0xc19bf174cf692694, 0xe49b69c19ef14ad2, 0xefbe4786384f25e3, 0x0fc19dc68b8cd5b5, 0x240ca1cc77ac9c65,
              0x2de92c6f592b0275, 0x4a7484aa6ea6e483, 0x5cb0a9dcbd41fbd4, 0x76f988da831153b5, 0x983e5152ee66dfab,
              0xa831c66d2db43210, 0xb00327c898fb213f, 0xbf597fc7beef0ee4, 0xc6e00bf33da88fc2, 0xd5a79147930aa725,
              0x06ca6351e003826f, 0x142929670a0e6e70, 0x27b70a8546d22ffc, 0x2e1b21385c26c926, 0x4d2c6dfc5ac42aed,
              0x53380d139d95b3df, 0x650a73548baf63de, 0x766a0abb3c77b2a8, 0x81c2c92e47edaee6, 0x92722c851482353b,
              0xa2bfe8a14cf10364, 0xa81a664bbc423001, 0xc24b8b70d0f89791, 0xc76c51a30654be30, 0xd192e819d6ef5218,
              0xd69906245565a910, 0xf40e35855771202a, 0x106aa07032bbd1b8, 0x19a4c116b8d2d0c8, 0x1e376c085141ab53,
              0x2748774cdf8eeb99, 0x34b0bcb5e19b48a8, 0x391c0cb3c5c95a63, 0x4ed8aa4ae3418acb, 0x5b9cca4f7763e373,
              0x682e6ff3d6b2b8a3, 0x748f82ee5defb2fc, 0x78a5636f43172f60, 0x84c87814a1f0ab72, 0x8cc702081a6439ec,
              0x90befffa23631e28, 0xa4506cebde82bde9, 0xbef9a3f7b2c67915, 0xc67178f2e372532b, 0xca273eceea26619c,
              0xd186b8c721c0c207, 0xeada7dd6cde0eb1e, 0xf57d4f7fee6ed178, 0x06f067aa72176fba, 0x0a637dc5a2c898a6,
              0x113f9804bef90dae, 0x1b710b35131c471b, 0x28db77f523047d84, 0x32caab7b40c72493, 0x3c9ebe0a15c9bebc,
              0x431d67c49c100d4c, 0x4cc5d4becb3e42b6, 0x597f299cfc657e2a, 0x5fcb6fab3ad6faec, 0x6c44198c4a475817]
```

4. Compression Function

   Now, we need to have a wider look at the hash function so that we can understand what is happening. First of all, we take 1024 bits of messages and divide the complete message in ‘n’ bits.



Now that we know the methodology to obtain the values of W for 0 to 79 and already have the values for K as well for all the rounds from 0 to 79 we can proceed ahead and see where and how we do put in these values for the computation of the hash.



In the image above we can see exactly what happens in each round and now that we have the values and formulas for each of the functions carried out we can perform the entire hashing process.

This, gives us the hashed output in SHA-512.

Code:

```
import binascii
import struct

initial_hash = (
    0x6a09e667f3bcc908,
    0xbb67ae8584caa73b,
    0x3c6ef372fe94f82b,
    0xa54ff53a5f1d36f1,
    0x510e527fade682d1,
    0x9b05688c2b3e6c1f,
    0x1f83d9abfb41bd6b,
    0x5be0cd19137e2179,
)

round_constants = (
    0x428a2f98d728ae22, 0x7137449123ef65cd, 0xb5c0fbcfec4d3b2f,
    0xe9b5dba58189dbbc, 0x3956c25bf348b538, 0x59f111f1b605d019,
    0x923f82a4af194f9b, 0xab1c5ed5da6d8118, 0xd807aa98a3030242,
    0x12835b0145706fbe, 0x243185be4ee4b28c, 0x550c7dc3d5ffb4e2,
    0x72be5d74f27b896f, 0x80deb1fe3b1696b1, 0x9bdc06a725c71235,
    0xc19bf174cf692694, 0xe49b69c19ef14ad2, 0xefbe4786384f25e3,
    0x0fc19dc68b8cd5b5, 0x240ca1cc77ac9c65, 0x2de92c6f592b0275,
    0x4a7484aa6ea6e483, 0x5cb0a9dcbd41fbd4, 0x76f988da831153b5,
```

```
    0x983e5152ee66dfab, 0xa831c66d2db43210, 0xb00327c898fb213f,
    0xbf597fc7beef0ee4, 0xc6e00bf33da88fc2, 0xd5a79147930aa725,
    0x06ca6351e003826f, 0x142929670a0e6e70, 0x27b70a8546d22ffc,
    0x2e1b21385c26c926, 0x4d2c6dfc5ac42aed, 0x53380d139d95b3df,
    0x650a73548baf63de, 0x766a0abb3c77b2a8, 0x81c2c92e47edaee6,
    0x92722c851482353b, 0xa2bfe8a14cf10364, 0xa81a664bbc423001,
    0xc24b8b70d0f89791, 0xc76c51a30654be30, 0xd192e819d6ef5218,
    0xd69906245565a910, 0xf40e35855771202a, 0x106aa07032bbd1b8,
    0x19a4c116b8d2d0c8, 0x1e376c085141ab53, 0x2748774cdf8eeb99,
    0x34b0bcb5e19b48a8, 0x391c0cb3c5c95a63, 0x4ed8aa4ae3418acb,
    0x5b9cca4f7763e373, 0x682e6ff3d6b2b8a3, 0x748f82ee5defb2fc,
    0x78a5636f43172f60, 0x84c87814a1f0ab72, 0x8cc702081a6439ec,
    0x90befffa23631e28, 0xa4506cebde82bde9, 0xbef9a3f7b2c67915,
    0xc67178f2e372532b, 0xca273eceea26619c, 0xd186b8c721c0c207,
    0xeada7dd6cde0eb1e, 0xf57d4f7fee6ed178, 0x06f067aa72176fba,
    0x0a637dc5a2c898a6, 0x113f9804bef90dae, 0x1b710b35131c471b,
    0x28db77f523047d84, 0x32caab7b40c72493, 0x3c9ebe0a15c9bebc,
    0x431d67c49c100d4c, 0x4cc5d4becb3e42b6, 0x597f299cfc657e2a,
    0x5fcb6fab3ad6faec, 0x6c44198c4a475817,
)


def _right_rotate(n: int, bits: int) -> int:
    return (n >> bits) | (n << (64 - bits)) & 0xFFFFFFFFFFFFFFFF


def sha512(message: str) -> str:
    if type(message) is not str:
        raise TypeError('Given message should be a string.')
    message_array = bytearray(message, encoding='utf-8')

    mdi = len(message_array) % 128
    padding_len = 119 - mdi if mdi < 112 else 247 - mdi
    ending = struct.pack('!Q', len(message_array) << 3)
    message_array.append(0x80)
    message_array.extend([0] * padding_len)
    message_array.extend(bytearray(ending))

    sha512_hash = list(initial_hash)
    for chunk_start in range(0, len(message_array), 128):
        chunk = message_array[chunk_start:chunk_start + 128]

        w = [0] * 80
        w[0:16] = struct.unpack('!16Q', chunk)

        for i in range(16, 80):
            s0 = (
                _right_rotate(w[i - 15], 1) ^
                _right_rotate(w[i - 15], 8) ^
```

```
                (w[i - 15] >> 7)
            )
            s1 = (
                _right_rotate(w[i - 2], 19) ^
                _right_rotate(w[i - 2], 61) ^
                (w[i - 2] >> 6)
            )
            w[i] = (w[i - 16] + s0 + w[i - 7] + s1) & 0xFFFFFFFFFFFFFFFF

        a, b, c, d, e, f, g, h = sha512_hash

        for i in range(80):
            sum1 = (
                _right_rotate(e, 14) ^
                _right_rotate(e, 18) ^
                _right_rotate(e, 41)
            )
            ch = (e & f) ^ (~e & g)
            temp1 = h + sum1 + ch + round_constants[i] + w[i]
            sum0 = (
                _right_rotate(a, 28) ^
                _right_rotate(a, 34) ^
                _right_rotate(a, 39)
            )
            maj = (a & b) ^ (a & c) ^ (b & c)
            temp2 = sum0 + maj

            h = g
            g = f
            f = e
            e = (d + temp1) & 0xFFFFFFFFFFFFFFFF
            d = c
            c = b
            b = a
            a = (temp1 + temp2) & 0xFFFFFFFFFFFFFFFF

        sha512_hash = [
            (x + y) & 0xFFFFFFFFFFFFFFFF
            for x, y in zip(sha512_hash, (a, b, c, d, e, f, g, h))
        ]

    return binascii.hexlify(
        b''.join(struct.pack('!Q', element) for element in sha512_hash),
    ).decode('utf-8')


if __name__ == "__main__":
    print("\n-- SHA512 --\n")
    print("Enter Message: ", end='')
```

```
    message = input()
    messageDigest = sha512(message)
    print("\nMessage Digest is:\n"+messageDigest)
```

We now illustrate with examples:

Say we wish to hash – ‘Password@1234’

```
PS C:\Users\marcus\Desktop\College\CNS-Lab-Archives\SHA512> py .\script.py

-- SHA512 --

Enter Message: Password@1234

Message Digest is:
1154e4485abba57f484c30a2ea543144178ff6a03da4503678a04ff8e808445709476e2e058a
f1c35afb1842eff5b7182d5d18fc3b98f185c47472520169af71
PS C:\Users\marcus\Desktop\College\CNS-Lab-Archives\SHA512>
```

Message Digest is:

1154e4485abba57f484c30a2ea543144178ff6a03da4503678a04ff8e8084457094
76e2e058af1c35afb1842eff5b7182d5d18fc3b98f185c47472520169af71

We take another example:

We now wish to hash ‘Password@123’

```
PS C:\Users\marcus\Desktop\College\CNS-Lab-Archives\SHA512> py .\script.py

-- SHA512 --

Enter Message: Password@123

Message Digest is:
e8be9807c3ec0bd1ce2a58e9cddd4a89966820aedc509830b62f05a226dd29a3764e4962acf2
8b4b87377b347b9395e1dd5bc87fed1e44f17315b89d7501836c
PS C:\Users\marcus\Desktop\College\CNS-Lab-Archives\SHA512>
```

Message Digest is:

e8be9807c3ec0bd1ce2a58e9cddd4a89966820aedc509830b62f05a226dd29a376
4e4962acf28b4b87377b347b9395e1dd5bc87fed1e44f17315b89d7501836c

Thus, we compute the hash. We also note how there is a significant difference in hash value even by just changing one character.

This property makes SHA-512 secure.

Thus, we demonstrated SHA-512 with examples.

Conclusion:

Thus, the SHA-512 algorithm was studied and demonstrated with the code.

The SHA-512 hashing algorithm is currently one of the best and secured hashing algorithms after hashes like MD5 and SHA-1 has been broken down. Due to their complicated nature it is not well accepted and SHA-256 is a general standard, but the industry is slowing moving towards this hashing algorithm.