NATIONAL INSTITUTE OF TECHNOLOGY PUDUCHERRY

(An Institution of National Importance under MoE, Govt. of India)

KARAIKAL – 609 609

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Roll Number: CS21B1005 **Name: Ankit Kujur**

Semester: 6th semester **Class:** CSE

Subject Code: CS1702 **Subject Name:** Network Security

Date: 29-04-25

SHA-512 Implementation Report

Abstract

This report details the implementation of the SHA-512 cryptographic hash function in Python, following the specifications outlined in FIPS PUB 180-4. The implementation processes an input string, applies padding, performs the compression function over 80 rounds, and produces a 512-bit hash. The results were validated against Python's hashlib library, confirming correctness. Challenges included handling bitwise operations and ensuring proper padding for variable-length inputs.

Table of Contents

1. Introduction

SHA-512 is a member of the SHA-2 family of cryptographic hash functions, designed by the National Security Agency (NSA) and standardized by NIST in FIPS PUB 180-4. It produces a 512-bit (64-byte) hash value, making it suitable for applications requiring high security, such as digital signatures, certificate generation, and password hashing. This report describes the implementation of SHA-512 in Python, adhering to the standard's specifications, and verifies its correctness by comparing outputs with a trusted library.

2. Objective

The primary objective was to implement the SHA-512 algorithm from scratch in Python, capable of processing arbitrary input strings and generating a 512-bit hash. The implementation was validated by comparing its output with the hashlib.sha512 function for the input string: *"This is the data to hash using SHA-512."*

3. Background Theory

SHA-512 is part of the SHA-2 family, which improves upon SHA-1 by offering larger hash sizes and enhanced security. It operates on 1024-bit message blocks, processes the input through 80 rounds of transformations, and outputs a 512-bit hash. Key components include:

- **Padding**: Appends a '1' bit, zero bits, and the message length to make the total length a multiple of 1024 bits.
- **Message Schedule**: Expands 16 words (64 bits each) into 80 words using bitwise operations.
- **Compression Function**: Updates eight 64-bit working variables (a to h) through 80 rounds, using functions like Ch, Maj, Sigma0, and Sigma1.
- **Initial Hash Values (H)**: Derived from the fractional parts of the square roots of the first eight prime numbers.
- **Round Constants (K)**: Derived from the cube roots of the first 80 prime numbers.

4. Methodology / Implementation Details

4.1 Tools and Language

The implementation was developed in Python 3.12.7, using the struct module for handling 64-bit integers, binascii for hexadecimal conversion, and hashlib for validation. No external cryptographic libraries were used for the core algorithm.

4.2 Implementation Steps

1. **Padding the Message:**
   - The pad_message function appends a single '1' bit (byte 0x80), followed by zero bytes, ensuring the total length is congruent to 896 modulo 1024 bits.
   - The original message length (in bits) is appended as a 128-bit big-endian integer.
2. **Initialization:**
   - Eight initial hash values (H0 to H7) and 80 round constants (K0 to K79) were defined as 64-bit integers, per the FIPS standard.
3. **Message Processing:**
   - The padded message is divided into 1024-bit (128-byte) blocks.
   - Each block is processed to create a message schedule (w[0...79]), where the first 16 words are directly from the block, and the remaining 64 are computed using sigma0 and sigma1 functions.
4. **Compression Function:**
   - For each block, eight working variables (a to h) are initialized with the current hash values.
   - Over 80 rounds, the variables are updated using Sigma0, Sigma1, Ch, and Maj functions, round constants (K[t]), and message schedule words (w[t]).
   - After 80 rounds, the hash values are updated by adding the final working variables.
5. **Final Hash:**
   - The eight 64-bit hash values are concatenated and converted to a hexadecimal string.

4.3 Code Implementation

The SHA-512 implementation consists of several key functions, organized to handle preprocessing, bitwise operations, and the compression process. Below are the primary components with code snippets and their roles:

Padding Function

The pad_message function prepares the input for processing by appending a '1' bit, zero bits, and the message length.

```
def pad_message(message_bytes):
    """Pad the input message according to SHA-512 specifications."""
    message_len_bits = len(message_bytes) * 8
    padding = b'\x80'  # Append a single '1' bit
    k_bits_needed = (896 - (message_len_bits + 1)) % 1024
    if k_bits_needed < 0:
        k_bits_needed += 1024
    k_bytes_needed = k_bits_needed // 8
    padding += b'\x00' * k_bytes_needed
    padding += struct.pack('>Q', 0)  # Upper 64 bits
    padding += struct.pack('>Q', message_len_bits)  # Lower 64 bits
    return message_bytes + padding
```

- **Role**: Ensures the message length is a multiple of 1024 bits, with space for a 128-bit length field.

Helper Functions

Eight helper functions perform bitwise and logical operations required for the message schedule and compression function:

1. **rotr(n, b)**: Right rotates a 64-bit integer by b bits.

```
def rotr(n, b):
    """Right rotate a 64-bit integer n by b bits."""
    return ((n >> b) | (n << (64 - b))) & 0xFFFFFFFFFFFFFFFF
```

2. **shr(n, b)**: Right shifts a 64-bit integer by b bits.

```
def shr(n, b):
    """Right shift a 64-bit integer n by b bits."""
    return (n >> b) & 0xFFFFFFFFFFFFFFFF
```

3. **Ch(x, y, z)**: Choose function, computes (x & y) ^ (~x & z).

```
def Ch(x, y, z):
    """Choose function."""
    return (x & y) ^ (~x & z)
```

4. **Maj(x, y, z)**: Majority function, computes (x & y) ^ (x & z) ^ (y & z).

```
def Maj(x, y, z):
    """Majority function."""
    return (x & y) ^ (x & z) ^ (y & z)
```

5. **Sigma0(x)**: Compression function transformation, combines rotations by 28, 34, and 39 bits.

```
def Sigma0(x):
    """As defined in FIPS PUB 180-4 section 4.1.3."""
    return rotr(x, 28) ^ rotr(x, 34) ^ rotr(x, 39)
```

6. **Sigma1(x)**: Compression function transformation, combines rotations by 14, 18, and 41 bits.

```python
def Sigma1(x):
    """As defined in FIPS PUB 180-4 section 4.1.3."""
    return rotr(x, 14) ^ rotr(x, 18) ^ rotr(x, 41)
```

7. **sigma0(x)**: Message schedule transformation, combines rotations by 1, 8, and a shift by 7 bits.

```python
def sigma0(x):
    """As defined in FIPS PUB 180-4 section 4.1.3."""
    return rotr(x, 1) ^ rotr(x, 8) ^ shr(x, 7)
```

8. **sigma1(x)**: Message schedule transformation, combines rotations by 19, 61, and a shift by 6 bits.

```python
def sigma1(x):
    """As defined in FIPS PUB 180-4 section 4.1.3."""
    return rotr(x, 19) ^ rotr(x, 61) ^ shr(x, 6)
```

- **Role**: These functions implement the bitwise operations specified in FIPS PUB 180-4, ensuring accurate transformations for 64-bit words.

Main SHA-512 Function

The calculate_sha512 function orchestrates the hashing process:

```python
def calculate_sha512(data_string):
    """Compute the SHA-512 hash of the input string."""
    # Encode input string to bytes
    message_bytes = data_string.encode('utf-8')

    # Pad the message
    padded_message = pad_message(message_bytes)

    # Initialize hash values
    h = list(H)

    # Process message in 1024-bit (128-byte) chunks
    block_size = 128
    num_blocks = len(padded_message) // block_size

    for i in range(num_blocks):
        # Get current block
        block_start = i * block_size
        block = padded_message[block_start: block_start + block_size]

        # Prepare message schedule (W[0...79])
        w = [0] * 80
        for t in range(16):
            offset = t * 8
            w[t] = struct.unpack('>Q', block[offset: offset + 8])[0]
```

```python
    # Extend the first 16 words
    for t in range(16, 80):
        s0 = sigma0(w[t-15])
        s1 = sigma1(w[t-2])
        w[t] = (w[t-16] + s0 + w[t-7] + s1) & 0xFFFFFFFFFFFFFFFF

    # Initialize working variables
    a, b, c, d, e, f, g, hh = h

    # Compression function (80 rounds)
    for t in range(80):
        S1 = Sigma1(e)
        ch = Ch(e, f, g)
        temp1 = (hh + S1 + ch + K[t] + w[t]) & 0xFFFFFFFFFFFFFFFF
        S0 = Sigma0(a)
        maj = Maj(a, b, c)
        temp2 = (S0 + maj) & 0xFFFFFFFFFFFFFFFF
        hh = g
        g = f
        f = e
        e = (d + temp1) & 0xFFFFFFFFFFFFFFFF
        d = c
        c = b
        b = a
        a = (temp1 + temp2) & 0xFFFFFFFFFFFFFFFF

    # Update hash values
    h[0] = (h[0] + a) & 0xFFFFFFFFFFFFFFFF
    h[1] = (h[1] + b) & 0xFFFFFFFFFFFFFFFF
    h[2] = (h[2] + c) & 0xFFFFFFFFFFFFFFFF
    h[3] = (h[3] + d) & 0xFFFFFFFFFFFFFFFF
    h[4] = (h[4] + e) & 0xFFFFFFFFFFFFFFFF
    h[5] = (h[5] + f) & 0xFFFFFFFFFFFFFFFF
    h[6] = (h[6] + g) & 0xFFFFFFFFFFFFFFFF
    h[7] = (h[7] + hh) & 0xFFFFFFFFFFFFFFFF

# Concatenate final hash
final_hash_bytes = b''.join(struct.pack('>Q', val) for val in h)
return binascii.hexlify(final_hash_bytes).decode('utf-8')
```

- **Role**: Encodes the input, pads it, processes each block through the message schedule and compression function, and formats the final hash.

5. Results

The implementation was tested with the input string: *"This is the data to hash using SHA-512."*. The output was:

- **SHA-512 Hash (from scratch)**:8fa60bf36ea065724612af56578778671569cb4256e69f12548e1bb4c4e40c5f1b5c92b2a9bab52c3e35aeb352c96f1bb49075db2e7855516e6417dc73fcf2dc
- **SHA-512 Hash (hashlib)**:8fa60bf36ea065724612af56578778671569cb4256e69f12548e1bb4c4e40c5f1b5c92b2a9bab52c3e35aeb352c96f1bb49075db2e7855516e6417dc73fcf2dc
- **Hash Length**: 128 characters (512 bits).
- **Results Match**: True.

The identical outputs confirm the implementation's correctness.