SHA-512: Secure Hash Algorithm 512-bit

SHA-512 is a cryptographic hash function that produces a 512-bit (64-byte) hash value. It's part of the SHA-2 family of hash functions, designed by the U.S. National Security Agency (NSA).

Key Characteristics

- **Input**: Can process messages up to 2^128 bits in length
- **Output**: 512-bit hash value
- **Block size**: 1024 bits
- **Word size**: 64 bits

Algorithm Steps

1. **Preprocessing**:
   - Pad the message:
     1. Append a single '1' bit to the message
     2. Add '0' bits until the length is 896 mod 1024
     3. Append the original message length as a 128-bit integer
   - Parse the padded message into 1024-bit blocks (M_1, M_2, ..., M_N)

2. **Initialize Hash Values**:
   - Set eight 64-bit words (H_0^0 to H_7^0) with specific initial values

3. **Process Message Blocks**:
   - For each 1024-bit block M_i (i = 1 to N):

     1. Prepare the message schedule (W_t):
        - Set W_0, W_1, ..., W_{15} as the 64-bit words of the current block
        - For t = 16 to 79: $W_t = \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16}$ where σ_0 and σ_1 are specialized functions

     2. Initialize working variables: $a = H_0^{i-1}, b = H_1^{i-1}, ..., h = H_7^{i-1}$

     3. Main loop (t = 0 to 79):
        - $T_1 = h + \Sigma_1(e) + \text{Ch}(e, f, g) + K_t + W_t$
        - $T_2 = \Sigma_0(a) + \text{Maj}(a, b, c)$
        - $h = g, g = f, f = e, e = d + T_1$
        - $d = c, c = b, b = a, a = T_1 + T_2$

        where Σ_0, Σ_1, Ch, and Maj are specialized functions

     4. Compute intermediate hash value: $H_0^i = a + H_0^{i-1}, H_1^i = b + H_1^{i-1}, ..., H_7^i = h + H_7^{i-1}$

4. **Produce Final Hash**:
   - Concatenate $H_0^N \parallel H_1^N \parallel ... \parallel H_7^N$ to form the 512-bit hash

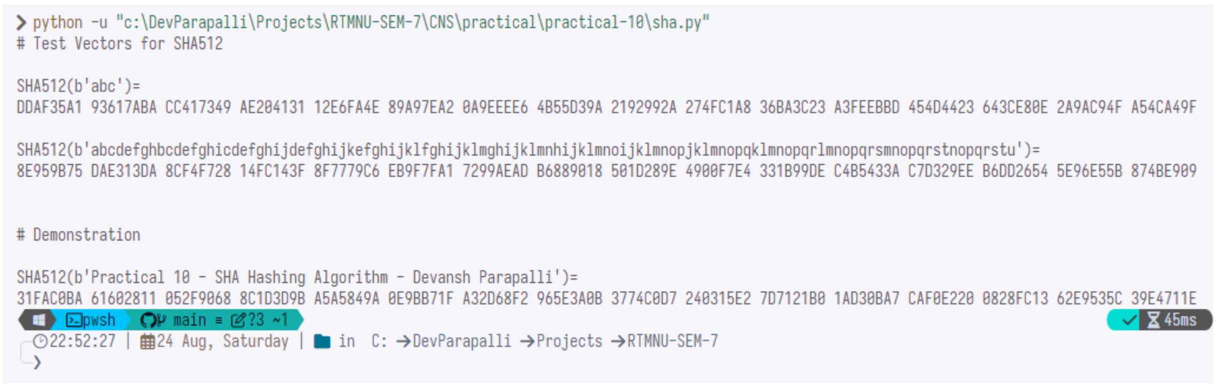
Key Features

- One-way function: computationally infeasible to reverse
- Deterministic: same input always produces the same output
- Designed to be collision-resistant
- Used in various security applications and protocols

Listing 1: sha.py

```python
class SHA512:
    K = [
        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
    ]

    # FIPS PUB 180-4 Section 5.3.5
    IV = [
        0x6a09e667f3bcc908, 0xbb67ae8584caa73b, 0x3c6ef372fe94f82b, 0xa54ff53a5f1d36f1,
        0x510e527fade682d1, 0x9b05688c2b3e6c1f, 0x1f83d9abfb41bd6b, 0x5be0cd19137e2179
    ]

    def __init__(self):
        self.hash_value = self.IV.copy()
        self.message = b""

    @staticmethod
    def _rotr(x: int, n: int) -> int:
        #Circular rotate x by n bits (aligned to 64-bit words) FIPS PUB 180-4 Section 2.2.2, Section 3.2
        return (x >> n) | (x << (64 - n))

    @staticmethod
    def _shr(x: int, n: int) -> int:
        #right shift x by n bits FIPS PUB 180-4 Section 2.2.2, Section 3.2
        return x >> n

    @staticmethod
    def _ch(x: int, y: int, z: int) -> int:
        # Ch function as given in FIPS PUB 180-4 Section 4.1.3 (4.2)
        return (x & y) ^ (~x & z)

    @staticmethod
    def _maj(x: int, y: int, z: int) -> int:
        # Maj function as given in FIPS PUB 180-4 Section 4.1.3 (4.3)
        return (x & y) ^ (x & z) ^ (y & z)

    @classmethod
    def _sigma0(cls, x: int) -> int:
        # SIGMA_0 function as given in FIPS PUB 180-4 Section 4.1.3 (4.4)
        return cls._rotr(x, 28) ^ cls._rotr(x, 34) ^ cls._rotr(x, 39)

    @classmethod
    def _sigma1(cls, x: int) -> int:
        # SIGMA_1 function as given in FIPS PUB 180-4 Section 4.1.3 (4.5)
        return cls._rotr(x, 14) ^ cls._rotr(x, 18) ^ cls._rotr(x, 41)

    @classmethod
    def _gamma0(cls, x: int) -> int:
        # sigma_0 function as given in FIPS PUB 180-4 Section 4.1.3 (4.6)
        return cls._rotr(x, 1) ^ cls._rotr(x, 8) ^ cls._shr(x, 7)

    @classmethod
    def _gamma1(cls, x: int) -> int:
        # sigma_1 function as given in FIPS PUB 180-4 Section 4.1.3 (4.7)
        return cls._rotr(x, 19) ^ cls._rotr(x, 61) ^ cls._shr(x, 6)

    def update(self, message: bytes) -> None:
        self.message += message

    def _pad(self) -> bytes:
        # FIPS PUB 180-4 Section 5.1.2
        length_of_message_bits = len(self.message) * 8
        ending = length_of_message_bits.to_bytes(16, "big")
        k = 0
        while (length_of_message_bits + 1 + k) % 1024 != 896:
            k += 1
        padding = 1 << k
        return self.message + padding.to_bytes((padding.bit_length() + 7) // 8, "big") + ending

    def _process_block(self, block: bytes) -> None:
        # FIPS PUB 180-4 Section 5.2.2
        message_blocks = [int.from_bytes(block[i:i+8], "big") for i in range(0, len(block), 8)]

        # FIPS PUB 180-4 Section 6.4.2
        W = message_blocks.copy()
        for t in range(16, 80):
            W.append((self._gamma1(W[t-2]) + W[t-7] + self._gamma0(W[t-15]) + W[t-16]) % 2**64)

        a, b, c, d, e, f, g, h = self.hash_value

        for t in range(80):
            t1 = (h + self._sigma1(e) + self._ch(e, f, g) + self.K[t] + W[t]) % 2**64
            t2 = (self._sigma0(a) + self._maj(a, b, c)) % 2**64

            h = g
            g = f
            f = e
            e = (d + t1) % 2**64
            d = c
            c = b
            b = a
            a = (t1 + t2) % 2**64

        self.hash_value = [(x + y) % 2**64 for x, y in zip(self.hash_value, [a, b, c, d, e, f, g, h])]

    def digest(self) -> bytes:
        padded_message = self._pad()
        blocks = [padded_message[i:i+128] for i in range(0, len(padded_message), 128)]

        for block in blocks:
            self._process_block(block)

        return b"".join(h.to_bytes(8, "big") for h in self.hash_value)

    def hexdigest(self) -> str:
        return self.digest().hex()

# Test Vectors for SHA512 (NIST)
#Link : https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Standards-and-Guidelines/documents/
examples/SHA512.pdf

print("# Test Vectors for SHA512")
_test_vector_0 = b"abc"
test_vector_0 = SHA512()
test_vector_0.update(_test_vector_0)
digest_0 = test_vector_0.hexdigest().upper()
print(f"\nSHA512({_test_vector_0})=\n{" ".join([digest_0[i:i+8] for i in range(0, len(digest_0),
8)])}" )
```

```python
_test_vector_1 =
b"abcdefghbcdefghicdefghijdefghijkefghijklfghijklmghijklmnhijklmnoijklmnopjklmnopqklmnopqrlmnopqrsmnopqrstnopqrstu"
test_vector_1 = SHA512()
test_vector_1.update(_test_vector_1)
digest_1 = test_vector_1.hexdigest().upper()
print(f"\nSHA512({_test_vector_1})=\n{" ".join([digest_1[i:i+8] for i in range(0, len(digest_1), 8)])}")

print("\n\n# Demonstration")
_demo = b"Practical 10 - SHA Hashing Algorithm - Devansh Parapalli"
demo = SHA512()
demo.update(_demo)
demo_digest = demo.hexdigest().upper()
print(f"\nSHA512({_demo})=\n{" ".join([demo_digest[i:i+8] for i in range(0, len(demo_digest), 8)])}")
```



```
> python -u "c:\DevParapalli\Projects\RTMNU-SEM-7\CNS\practical\practical-10\sha.py"
# Test Vectors for SHA512

SHA512(b'abc')=
DDAF35A1 93617ABA CC417349 AE204131 12E6FA4E 89A97EA2 0A9EEEE6 4B55D39A 2192992A 274FC1A8 36BA3C23 A3FEEBBD 454D4423 643CE80E 2A9AC94F A54CA49F

SHA512(b'abcdefghbcdefghicdefghijdefghijkefghijklfghijklmghijklmnhijklmnoijklmnopjklmnopqklmnopqrlmnopqrsmnopqrstnopqrstu')=
8E959B75 DAE313DA 8CF4F728 14FC143F 8F7779C6 EB9F7FA1 7299AEAD B6889018 501D289E 4900F7E4 331B99DE C4B5433A C7D329EE B6DD2654 5E96E55B 874BE909


# Demonstration

SHA512(b'Practical 10 - SHA Hashing Algorithm - Devansh Parapalli')=
31FAC0BA 61602811 052F9068 8C1D3D9B A5A5849A 0E9BB71F A32D68F2 965E3A0B 3774C0D7 240315E2 7D7121B0 1AD30BA7 CAF0E220 0828FC13 62E9535C 39E4711E
  pwsh   main ≡ ?3 ~1                                                                                                      ✓ 45ms
 22:52:27 | 24 Aug, Saturday |  in  C: →DevParapalli →Projects →RTMNU-SEM-7
 >
```

Figure 1: Output