SHA-256 STEP REDUCTION AND COLLISION ANALYSIS

MIHIR GUPTA
mihirgupta292@gmail.com

1. Introduction

A hash function is a mathematical algorithm that transforms an arbitrary input into a fixed-size string of bytes, known as the hash value or digest. Hash functions are designed to be fast and produce unique outputs for unique inputs, ensuring data integrity. However, when two different inputs produce the same hash value, it is known as a hash collision, which can compromise the security of the hash function. For instance, in cryptographic hash functions used for storing passwords, a hash collision could allow an attacker to gain access using a different password that produces the same hash as the original, compromising the security of the system.

Given the critical role of hash functions in securing data, the robustness of these functions—particularly their resistance to collisions—has been a focal point in cryptography. SHA-256, a member of the SHA-2 family, is one such function widely used in securing data across various applications. However, the resilience of hash functions like SHA-256 is critically dependent on their resistance to collision attacks. This paper explores collision attacks on step-reduced versions of SHA-256, focusing on 20-, 21-, 23-, and 47-step reductions. We describe the methodology for finding collisions in these reduced versions, emphasizing the mathematical structures that facilitate such attacks.

2. Mathematical Description of SHA-256

SHA-256 is a cryptographic hash function that processes an input message by dividing it into 512-bit blocks. Each block undergoes 64 steps of processing within a compression function. The function maintains an internal state consisting of eight 32-bit variables A, B, C, D, E, F, G, and H, which are updated at each step according to the following equations:

$$
\begin{aligned}
A_{i+1} &= \Sigma_0(A_i) + \mathrm{Maj}(A_i, B_i, C_i) + \Sigma_1(E_i) + \mathrm{Ch}(E_i, F_i, G_i) + H_i + K_i + W_i, \\
B_{i+1} &= A_i, \\
C_{i+1} &= B_i, \\
D_{i+1} &= C_i, \\
E_{i+1} &= \Sigma_1(E_i) + \mathrm{Ch}(E_i, F_i, G_i) + H_i + K_i + W_i + D_i, \\
F_{i+1} &= E_i, \\
G_{i+1} &= F_i, \\
H_{i+1} &= G_i.
\end{aligned}
$$

Date: July 2024.

Here, Σ_0 and Σ_1 are non-linear functions defined as:

$$\Sigma_0(X) = \text{ROTR}^2(X) \oplus \text{ROTR}^{13}(X) \oplus \text{ROTR}^{22}(X),$$
$$\Sigma_1(X) = \text{ROTR}^6(X) \oplus \text{ROTR}^{11}(X) \oplus \text{ROTR}^{25}(X),$$

where $\text{ROTR}^n(X)$ represents the right rotation of the 32-bit word X by n bits.

The rotation and shift operations are used to mix the bits in the words, adding complexity and diffusion to the hash function.

Right Rotation (ROTR): The operation $\text{ROTR}^n(X)$ rotates the 32-bit word X to the right by n bits. For example, if X is '10011010' and $n = 3$, then $\text{ROTR}^3(X)$ would result in '01010011'.

Right Shift (SHR): The operation $\text{SHR}^n(X)$ shifts the 32-bit word X to the right by n bits, filling the leftmost n bits with zeros. For example, if X is '10011010' and $n = 3$, then $\text{SHR}^3(X)$ would result in '00010011'.

The functions Maj and Ch are defined as:

$$\text{Maj}(X, Y, Z) = (X \wedge Y) \oplus (X \wedge Z) \oplus (Y \wedge Z),$$
$$\text{Ch}(X, Y, Z) = (X \wedge Y) \oplus (\neg X \wedge Z).$$

Majority (Maj): This function returns the majority bit of its three inputs:

$$\text{Maj}(X, Y, Z) = (X \wedge Y) \oplus (X \wedge Z) \oplus (Y \wedge Z)$$

This means that each bit of the output is '1' if at least two of the corresponding bits in X, Y, and Z are '1'.

Choice (Ch): This function selects bits from Y or Z, based on the value of X:

$$\text{Ch}(X, Y, Z) = (X \wedge Y) \oplus (\neg X \wedge Z)$$

If a bit in X is '1', the corresponding bit from Y is chosen; otherwise, the bit from Z is chosen.

The message block M is expanded into 64 words $W_0, W_1, \ldots, W_{63}$, where the first 16 words are directly taken from the message, and the remaining words are computed as:

$$W_i = \sigma_1(W_{i-2}) + W_{i-7} + \sigma_0(W_{i-15}) + W_{i-16},$$

with the functions σ_0 and σ_1 defined as:

$$\sigma_0(X) = \text{ROTR}^7(X) \oplus \text{ROTR}^{18}(X) \oplus \text{SHR}^3(X),$$
$$\sigma_1(X) = \text{ROTR}^{17}(X) \oplus \text{ROTR}^{19}(X) \oplus \text{SHR}^{10}(X),$$

where $\text{SHR}^n(X)$ denotes the right shift of X by n bits.

The message block is divided into 16 words initially, but the schedule W extends this to 64 words using a combination of the original message words and the σ_0 and σ_1 functions. This ensures that the data from the original message is mixed and diffused throughout the hash computation.

The constants K_i are 64 unique, predetermined values derived from the cube roots of the first 64 prime numbers, used to provide additional entropy in each iteration step of the SHA-256 compression function. These constants are essential to the security of SHA-256, introducing

fixed, but complex, numbers into the hash computation, which helps in preventing attacks like preimage and collision attacks.

We have attached Python implementation of SHA-256 in Listing 1, and a few example executions in Listing 2.

3. Types of Collision Attacks

Different types of collision attacks are used to exploit weaknesses in the hash function's ability to produce unique outputs for different inputs. The three primary types of collision attacks discussed in this paper are:

3.1. **Collision Attack.** This is the most basic type of attack where the goal is to find two different messages, M_1 and M_2, such that they produce the same hash value with the same initial value:

$$H(M_1, h_0) = H(M_2, h_0)$$

Here, h_0 is the initial chaining value.

3.2. **Semi-Free Start Collision Attack.** In this type of attack, the goal is to find two different messages, M_1 and M_2, and a specific initial hash value h_0^*, such that:

$$H(M_1, h_0^*) = H(M_2, h_0^*)$$

Unlike a full collision attack, we can choose a different initial value h_0^* for the hash function.

3.3. **Near Collision Attack.** This type of attack is focused on finding two different messages such that their hash values are close but not exactly the same. The difference between the hash values is small, typically measured in the number of differing bits (Hamming distance). For example, in a near collision with a Hamming distance of 15 bits:

$$\text{Hamming distance}(H(M_1, h_0), H(M_2, h_0)) = 15$$

Near collisions are often used to demonstrate partial weaknesses in the hash function.

4. Technique for Collision Creation

In this section we present steps from [1] to find collisions in 20 step-reduced SHA-256.

Differences mentioned below are subtractions mod 2^{32} differences. We use the following notations:

$$\Delta X = X' - X, \quad X \in \{A, B, C, D, E, F, G, H, W, m\}$$
$$\Delta\text{Maj}_i(\Delta a, \Delta b, \Delta c) = \text{Maj}(A_i + \Delta a, B_i + \Delta b, C_i + \Delta c) - \text{Maj}(A_i, B_i, C_i)$$
$$\Delta\text{Ch}_i(\Delta e, \Delta f, \Delta g) = \text{Ch}(E_i + \Delta e, F_i + \Delta f, G_i + \Delta g) - \text{Ch}(E_i, F_i, G_i)$$
$$\Delta\Sigma_0(A_i) = \Sigma_0(A_i') - \Sigma_0(A_i)$$
$$\Delta\Sigma_1(E_i) = \Sigma_1(E_i') - \Sigma_1(E_i)$$
$$\Delta\sigma_0(m_i) = \sigma_0(m_i') - \sigma_0(m_i)$$
$$\Delta\sigma_1(m_i) = \sigma_1(m_i') - \sigma_1(m_i)$$

We introduce a disturbance at step i and aim to correct the differences in the subsequent 8 steps. The following differential pattern is utilized:

Table 1. A 9-step differential for SHA-256.

Step	ΔA	ΔB	ΔC	ΔD	ΔE	ΔF	ΔG	ΔH	ΔW
i	0	0	0	0	0	0	0	0	1
$i+1$	1	0	0	0	1	0	0	0	δ_1
$i+2$	0	1	0	0	-1	1	0	0	δ_2
$i+3$	0	0	1	0	0	-1	1	0	δ_3
$i+4$	0	0	0	1	0	0	-1	1	0
$i+5$	0	0	0	0	1	0	0	-1	0
$i+6$	0	0	0	0	0	1	0	0	0
$i+7$	0	0	0	0	0	0	1	0	0
$i+8$	0	0	0	0	0	0	0	1	δ_4
$i+9$	0	0	0	0	0	0	0	0	0

As shown in the above table (column ΔW), the perturbations from ΔA through ΔH and the initial ΔW are fixed. The other perturbations remain to be solved for.

4.1. **Conditions for Local Collisions.** Focusing on A_{i+1} and E_{i+1}:

$$\Delta A_{i+1} - \Delta E_{i+1} = \Delta\Sigma_0(A_i) + \Delta\mathrm{Maj}_i(\Delta A_i, \Delta B_i, \Delta C_i) - \Delta D_i, \tag{4.1}$$

$$\Delta E_{i+1} = \Delta\Sigma_1(E_i) + \Delta\mathrm{Ch}_i(\Delta E_i, \Delta F_i, \Delta G_i) + \Delta H_i + \Delta D_i + \Delta W_i. \tag{4.2}$$

When $\Delta A_i = \Delta B_i = \Delta C_i = 0$, $\Delta\mathrm{Maj}_i(0,0,0) = 0$. Similarly, when $\Delta E_i = \Delta F_i = \Delta G_i = 0$, $\Delta\mathrm{Ch}_i(0,0,0) = 0$.

Fixing the differences in A and E (as shown in the table) means that B, C, D, F, G, and H can only inherit values from A and E. This results in equations involving δ_i and the values of A_i or E_i at each step.

Step $i+1$. With $\Delta D_i = 0$, $\Delta H_i = 0$, $\Delta\Sigma_0(A_i) = 0$, and $\Delta\Sigma_1(E_i) = 0$, we require $\Delta A_{i+1} = 1$ and $\Delta E_{i+1} = 1$, leading to:

$$\Delta W_i = 1 \tag{4.3}$$

Step $i+2$. Given $\Delta D_{i+1} = 0$ and $\Delta H_{i+1} = 0$, we need $\Delta A_{i+2} = 0$ and $\Delta E_{i+2} = -1$, while ensuring $\Delta\Sigma_0(A_{i+1}) = 1$. Thus:

$$\Delta\mathrm{Maj}_{i+1}(1,0,0) = 0 \tag{4.4}$$

$$\Delta W_{i+1} = -1 - \Delta\mathrm{Ch}_{i+1}(1,0,0) - \Delta\Sigma_1(E_{i+1}) \tag{4.5}$$

$$\Delta\Sigma_0(A_{i+1}) = 1 \tag{4.6}$$

Step $i+3$. With $\Delta D_{i+2} = 0$, $\Delta H_{i+2} = 0$, and $\Delta\Sigma_0(A_{i+2}) = 0$, we require $\Delta A_{i+3} = 0$ and $\Delta E_{i+3} = 0$, leading to:

$$\Delta\mathrm{Maj}_{i+2}(0,1,0) = 0 \tag{4.7}$$

$$\Delta W_{i+2} = -\Delta\Sigma_1(E_{i+2}) - \Delta\mathrm{Ch}_{i+2}(-1,1,0) \tag{4.8}$$

Step $i+4$. With $\Delta D_{i+3} = 0$, $\Delta H_{i+3} = 0$, $\Delta\Sigma_0(A_{i+3}) = 0$, and $\Delta\Sigma_1(E_{i+3}) = 0$, we require $\Delta A_{i+4} = 0$ and $\Delta E_{i+4} = 0$, resulting in:

$$\Delta\mathrm{Maj}_{i+3}(0,0,1) = 0 \tag{4.9}$$

$$\Delta W_{i+3} = -\Delta\mathrm{Ch}_{i+3}(0,-1,1) \tag{4.10}$$

Step $i+5$. With $\Delta D_{i+4} = 1$, $\Delta H_{i+4} = 1$, $\Delta\Sigma_0(A_{i+4}) = 0$, and $\Delta\Sigma_1(E_{i+4}) = 0$, we require $\Delta A_{i+5} = 0$ and $\Delta E_{i+5} = 1$, leading to:

$$\Delta\mathrm{Ch}_{i+4}(0,0,-1) = -1 \tag{4.11}$$

Step $i+6$. With $\Delta D_{i+5} = 0$, $\Delta H_{i+5} = -1$, and $\Delta\Sigma_0(A_{i+5}) = 0$, we require $\Delta A_{i+6} = 0$ and $\Delta E_{i+6} = 0$, while ensuring $\Delta\Sigma_0(E_{i+5}) = 1$. This leads to:

$$\Delta\mathrm{Ch}_{i+5}(1,0,0) = 0 \tag{4.12}$$

$$\Delta\Sigma_1(E_{i+5}) = 1 \tag{4.13}$$

Step $i+7$. With $\Delta D_{i+6} = 0$, $\Delta H_{i+6} = 0$, $\Delta\Sigma_0(A_{i+6}) = 0$, and $\Delta\Sigma_1(E_{i+6}) = 0$, we require $\Delta A_{i+7} = 0$ and $\Delta E_{i+7} = 0$, leading to:

$$\Delta\mathrm{Ch}_{i+6}(0,1,0) = 0 \tag{4.14}$$

Step $i+8$. With $\Delta D_{i+7} = 0$, $\Delta H_{i+7} = 0$, $\Delta\Sigma_0(A_{i+7}) = 0$, and $\Delta\Sigma_1(E_{i+7}) = 0$, we require $\Delta A_{i+8} = 0$ and $\Delta E_{i+8} = 0$, leading to:

$$\Delta\mathrm{Ch}_{i+7}(0,0,1) = 0 \tag{4.15}$$

Step $i+9$. With $\Delta D_{i+8} = 0$, $\Delta H_{i+8} = 1$, $\Delta\Sigma_0(A_{i+8}) = 0$, and $\Delta\Sigma_1(E_{i+8}) = 0$, we require $\Delta A_{i+9} = 0$ and $\Delta E_{i+9} = 0$, leading to:

$$\Delta W_{i+8} = -1 \tag{4.16}$$

4.2. **Solving the Equations.** First, consider equations (4.6) and (4.13). Given that $\Delta A_{i+1} = \Delta E_{i+5} = 1$, we require the functions $\Delta\Sigma_0(A_{i+1})$ and $\Delta\Sigma_1(E_{i+5})$ to preserve this difference of 1:

$$\Sigma_0(A_{i+1} + 1) - \Sigma_0(A_{i+1}) = 1, \tag{4.17}$$

$$\Sigma_1(E_{i+5} + 1) - \Sigma_1(E_{i+5}) = 1. \tag{4.18}$$

The only solution to these equations is $A_{i+1} = E_{i+5} = -1$, hence:

$$A_{i+1} = -1, \quad A'_{i+1} = 0, \tag{4.19}$$

$$E_{i+5} = -1, \quad E'_{i+5} = 0. \tag{4.20}$$

Next, consider the function $\Delta\text{Maj}_i = \text{Maj}(A'_i, B'_i, C'_i) - \text{Maj}(A_i, B_i, C_i)$. Assume that $B'_i = B_i$, $C'_i = C_i$, and A_i and A'_i differ in all bits, i.e., $A_i \oplus A'_i = 0xffffffff$. Then:

$$\Delta\text{Maj}_i = 0 \quad \text{if and only if} \quad B_i = C_i \tag{4.21}$$

Thus, from (4.4), we infer $B_{i+1} = C_{i+1}$, implying $A_i = A_{i-1}$. Similarly, from (4.7) and (4.9), we deduce:

$$A_{i-1} = A_i = A_{i+2} = A_{i+3} \tag{4.22}$$

For ΔCh_i, assume $F'_i = F_i$, $G'_i = G_i$, and E_i and E'_i differ in all bits. Then:

$$\Delta\text{Ch}_i = 0 \quad \text{if and only if} \quad F_i = G_i \tag{4.23}$$

Consequently, from (4.12) and the result in (4.20), we deduce $F_{i+5} = G_{i+5}$, leading to:

$$E_{i+4} = E_{i+3} \tag{4.24}$$

Solving (4.14) requires slightly different reasoning. If $E_{i+6} = E'_{i+6}$, $G_{i+6} = G'_{i+6}$, and F_{i+6} and F'_{i+6} differ in every bit (which they do, as shown in (4.20)), then:

$$\Delta\text{Ch}_{i+6} = 0 \quad \text{if and only if} \quad E_{i+6} = 0. \tag{4.25}$$

Similarly, from (4.15), we get:

$$E_{i+7} = -1 \tag{4.26}$$

The final condition is given by (4.11):

$$\Delta\text{Ch}_{i+4} = \text{Ch}(E_{i+4}, F_{i+4}, G'_{i+4}) - \text{Ch}(E_{i+4}, F_{i+4}, G_{i+4}) = -1 \tag{4.27}$$

Given that the words E_{i+4}, F_{i+4}, and G_{i+4} satisfy the earlier conditions, there are no degrees of freedom left to control the solution of this equation precisely. Thus, we estimate the probability that this condition holds. It holds if and only if E_{i+4} has 0's in the bits where G'_{i+4} and G_{i+4} differ. The difference between G'_{i+4} and G_{i+4} may be in the last i bits, where $1 \leq i \leq 32$, and these bits are uniquely determined. Therefore, the probability is:

$$\sum_{i=1}^{32} P\{\text{Last } i \text{ bits of } E_{i+4} \text{ are zero}\} \times P\{\text{Difference in exactly } i \text{ last bits}\} = \sum_{i=1}^{32} \frac{1}{2^i} \times \frac{1}{2^i} \approx \frac{1}{3}.$$

Thus, the overall probability of this differential is $\frac{1}{3} = 2^{-1.58}$. The differences in the message words of the differential are as follows:

$$\delta_1 = -1 - \Delta\mathrm{Ch}_{i+1}(1,0,0) - \Delta\Sigma_1(E_{i+1}),$$

$$\delta_2 = -\Delta\Sigma_1(E_{i+2}) - \Delta\mathrm{Ch}_{i+2}(-1,1,0),$$

$$\delta_3 = -\Delta\mathrm{Ch}_{i+3}(0,-1,1),$$

$$\delta_4 = -1.$$

Notice that the condition (4.22) implies that $A_i = B_i$ must hold.

We observe that the words m_5, m_6, m_7, m_8, and m_{13} are each used only once in the first 20 steps of SHA-256 (Please see Table 2), meaning they are not used to compute the expanded words beyond step 14: W_{15}, W_{16}, W_{17}, W_{18}, and W_{19}. For the 20-step collision attack, i is set to 5, so $i + 9$ equals 14. Because the perturbations introduced in Table 1 do not affect steps beyond 14, as seen in Table 2, this differential allows us to find a collision. Therefore, a collision can be found for the 20-step reduced SHA-256 and the probability of this collision occurring is $2^{-1.58}$.

5. 21-step and 23-step reduced SHA-256 collision

21-Step Collision. The 21-step reduced SHA-256 introduces an additional level of complexity by extending the differential attack by one more step.

Differential Construction. For the 21-step collision, differences are introduced in the message words m_6, m_7, m_8, m_9, and m_{14}. m_6, m_7, and m_8 are only used once in the first 21 steps hence the message expansion in the first 21 steps is irrelevant with respect to these words. But as seen in Table 2, m_9 and m_{14} are used in ≥ 1 in the first 21 steps. So, we have to find a $m_9, m'_9, m_{14}, and\ m'_{14}$ such that the message expansion will not introduce any differences past the fourteenth step. From Table 2 we see that m_9 and m_{14} are used in W_{16}, W_{18}, and W_{20} giving us the following equations:

$$\Delta W_{16} = \Delta\sigma_1(m_{14}) + \Delta m_9 + \Delta\sigma_0(m_1) + \Delta m_0 = 0$$
$$\Delta W_{17} = \Delta\sigma_1(m_{15}) + \Delta m_{10} + \Delta\sigma_0(m_2) + \Delta m_1 = 0$$
$$\Delta W_{18} = \Delta\sigma_1(W_{16}) + \Delta m_{11} + \Delta\sigma_0(m_3) + \Delta m_2 = 0$$
$$\Delta W_{19} = \Delta\sigma_1(W_{17}) + \Delta m_{12} + \Delta\sigma_0(m_4) + \Delta m_3 = 0$$
$$\Delta W_{20} = \Delta\sigma_1(W_{18}) + \Delta m_{13} + \Delta\sigma_0(m_5) + \Delta m_4 = 0$$

If $m'_i = m_i$ ($W'_i = W_i$), then $\Delta\sigma_0(m_i) = 0$ ($\Delta\sigma_0(W_i) = 0$). This implies $\Delta W_{17} = \Delta W_{19} = 0$. If $\Delta W_{16} = 0$, then $\Delta W_{18} = \Delta W_{20} = 0$. This leads to the equation:

$$\Delta\sigma_1(m_{14}) + \Delta m_9 = 0$$

[1] found that the probability of finding a 21-step collision using the above methodology is 2^{-19}.

Execution of the Attack. The attack follows a similar procedure as the 20-step attack. By step 21, the differentials cancel out, leading to a collision with a probability of approximately 2^{-19}.

23-Step Semi-Free Start Collision. The 23-step reduced SHA-256 attack operates under semi-free start conditions, where we have control over the initial state h_0.

The semi-free start collision attack involves introducing differences in the message words m_9, m_{10}, m_{11}, and m_{12}. The differential propagation for this attack is governed by equations derived in a manner similar to 21-steps reduced SHA-256. The goal is to ensure that by the 23rd step, the differences cancel out, resulting in a collision with a probability of approximately 2^{-21} [1]. [1] also shows a semi-free start near collision with hamming distance of 15 bits with probability of 2^{-34} for 25 step-reduced SHA-256.

Second-Order Differential Collision. [2] using the theory of higher order differentials shows a collision for 47-step reduced SHA-256 with 2^{46} complexity. The 47-step attack involves a forward phase and a backward phase. In the backward phase, differentials are applied starting from step 22 back. The forward phase applies differentials from step 22 to step 47.

Key components of this attack are:

(1) **Linearization**: The SHA-256 compression function is approximated by linearizing the modular additions and replacing them by the XOR operations.
(2) **Boolean Function Approximation**: Boolean functions f_0 (i.e. Maj) and f_1 (i.e. Ch) in SHA-256 are approximated by the 0-function, except in the jth bit, where either $\Delta A[j] = \Delta B[j] = \Delta C[j] = 1$ or $\Delta F[j] = \Delta G[j] = 1$.

6. Conclusion

This paper details various step-reduced collision attacks on SHA-256 hash. These attacks highlight the vulnerability of the compression function under certain conditions. They reveal

potential weaknesses in the SHA-2 family of hash functions and highlights the importance of ongoing evaluation and the need for robust cryptographic standards.

References

[1] I. Nikolić and A. Biryukov, "Collisions for step-reduced sha-256," in *Fast Software Encryption: 15th International Workshop, FSE 2008, Lausanne, Switzerland, February 10-13, 2008, Revised Selected Papers 15*. Springer, 2008, pp. 1–15.

[2] A. Biryukov, M. Lamberger, F. Mendel, and I. Nikolić, "Second-order differential collisions for reduced sha-256," in *Advances in Cryptology–ASIACRYPT 2011: 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings 17*. Springer, 2011, pp. 270–287.

Appendix

Listing 1. Python code for calculating SHA-256 hash.

```
"""This Python module is an implementation of the SHA-256 algorithm.
From https://github.com/keanemind/Python-SHA-256"""

K = [
    0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
    0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
    0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
    0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
    0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
    0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
    0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
    0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2
]

def generate_hash(message: bytearray) -> bytearray:
    """Return a SHA-256 hash from the message passed.
    The argument should be a bytes, bytearray, or
    string object."""

    if isinstance(message, str):
        message = bytearray(message, 'ascii')
    elif isinstance(message, bytes):
        message = bytearray(message)
    elif not isinstance(message, bytearray):
        raise TypeError

    # Padding
    length = len(message) * 8 # len(message) is number of BYTES!!!
    message.append(0x80)
    while (len(message) * 8 + 64) % 512 != 0:
        message.append(0x00)

    message += length.to_bytes(8, 'big') # pad to 8 bytes or 64 bits

    assert (len(message) * 8) % 512 == 0, "Padding did not complete properly!"

    # Parsing
    blocks = [] # contains 512-bit chunks of message
    for i in range(0, len(message), 64): # 64 bytes is 512 bits
        blocks.append(message[i:i+64])

    # Setting Initial Hash Value
    h0 = 0x6a09e667
    h1 = 0xbb67ae85
    h2 = 0x3c6ef372
    h3 = 0xa54ff53a
    h5 = 0x9b05688c
```

```
    h4 = 0x510e527f
    h6 = 0x1f83d9ab
    h7 = 0x5be0cd19

    # SHA-256 Hash Computation
    for message_block in blocks:
        # Prepare message schedule
        message_schedule = []
        for t in range(0, 64):
            if t <= 15:
                # adds the t'th 32 bit word of the block,
                # starting from leftmost word
                # 4 bytes at a time
                message_schedule.append(bytes(message_block[t*4:(t*4)+4]))
            else:
                term1 = _sigma1(int.from_bytes(message_schedule[t-2], 'big'))
                term2 = int.from_bytes(message_schedule[t-7], 'big')
                term3 = _sigma0(int.from_bytes(message_schedule[t-15], 'big'))
                term4 = int.from_bytes(message_schedule[t-16], 'big')

                # append a 4-byte byte object
                schedule = ((term1 + term2 + term3 + term4) % 2**32).to_bytes(4, 'big')
                message_schedule.append(schedule)

        assert len(message_schedule) == 64

        # Initialize working variables
        a = h0
        b = h1
        c = h2
        d = h3
        e = h4
        f = h5
        g = h6
        h = h7

        # Iterate for t=0 to 63
        for t in range(64):
            t1 = ((h + _capsigma1(e) + _ch(e, f, g) + K[t] +
                   int.from_bytes(message_schedule[t], 'big')) % 2**32)

            t2 = (_capsigma0(a) + _maj(a, b, c)) % 2**32

            h = g
            g = f
            f = e
            e = (d + t1) % 2**32
            d = c
            c = b
            b = a
            a = (t1 + t2) % 2**32

        # Compute intermediate hash value
        h0 = (h0 + a) % 2**32
        h1 = (h1 + b) % 2**32
        h2 = (h2 + c) % 2**32
        h3 = (h3 + d) % 2**32
        h4 = (h4 + e) % 2**32
        h5 = (h5 + f) % 2**32
        h6 = (h6 + g) % 2**32
        h7 = (h7 + h) % 2**32

    return ((h0).to_bytes(4, 'big') + (h1).to_bytes(4, 'big') +
            (h2).to_bytes(4, 'big') + (h3).to_bytes(4, 'big') +
            (h4).to_bytes(4, 'big') + (h5).to_bytes(4, 'big') +
            (h6).to_bytes(4, 'big') + (h7).to_bytes(4, 'big'))

def _sigma0(num: int):
```

```
    """As defined in the specification."""
    num = (_rotate_right(num, 7) ^
           _rotate_right(num, 18) ^
           (num >> 3))
    return num

def _sigma1(num: int):
    """As defined in the specification."""
    num = (_rotate_right(num, 17) ^
           _rotate_right(num, 19) ^
           (num >> 10))
    return num

def _capsigma0(num: int):
    """As defined in the specification."""
    num = (_rotate_right(num, 2) ^
           _rotate_right(num, 13) ^
           _rotate_right(num, 22))
    return num

def _capsigma1(num: int):
    """As defined in the specification."""
    num = (_rotate_right(num, 6) ^
           _rotate_right(num, 11) ^
           _rotate_right(num, 25))
    return num

def _ch(x: int, y: int, z: int):
    """As defined in the specification."""
    return (x & y) ^ (~x & z)

def _maj(x: int, y: int, z: int):
    """As defined in the specification."""
    return (x & y) ^ (x & z) ^ (y & z)

def _rotate_right(num: int, shift: int, size: int = 32):
    """Rotate an integer right."""
    return (num >> shift) | (num << size - shift)

if __name__ == "__main__":
    print(generate_hash("Hello").hex())
```

Listing 2. Example SHA-256 hashes

```
# Examples
print(generate_hash("Hello World").hex())
# a591a6d40bf420404a011733cfb7b190d62c65bf0bcda32b57b277d9ad9f146e
print(generate_hash("Hello").hex())
# 185f8db32271fe25f561a6fc938b2e264306ec304eda518007d1764826381969
print(generate_hash("Hella").hex())
# 62197ce641a17c96cf39ed51e72e86fd486f3de7b0135f8b8f0f7364dc010e54
```

Table 2. Message expansion of SHA-256. There is 'x' in the intersection of row with index i and column with index j if W_i uses m_j.

W	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	x															
1		x														
2			x													
3				x												
4					x											
5						x										
6							x									
7								x								
8									x							
9										x						
10											x					
11												x				
12													x			
13														x		
14															x	
15																x
16	x	x								x					x	
17		x	x								x					x
18	x	x	x							x		x			x	
19		x	x	x	x						x		x			x
20	x	x	x	x	x	x				x		x		x	x	
21		x	x	x	x	x	x				x		x		x	x
22	x	x	x	x	x	x	x	x		x		x		x	x	x