| Component | Links |
|---|---|
| ATSHA204A | https://en.wikipedia.org/wiki/SHA-2 |
| | https://www.elektronik-kompendium.de/sites/net/1910161.htm |
| | http://www.iwar.org.uk/comsec/resources/cipher/sha256-384-512.pdf |

The ATSHA204 uses the Secure Hash Algorithm Version 2 with a bit length of 256 (SHA256). SHA 256 is used in several Crypto Currencies like Bitcoin and Security Protocols like DNSSEC.

From http://www.iwar.org.uk/comsec/resources/cipher/sha256-384-512.pdf :

## 1.0 Overview

SHA-256 operates in the manner of MD4, MD5, and SHA-1: The message to be hashed is first

- padded with its length in such a way that the result is a multiple of 521 bits long, and then
- Parsed into 512-bit message blocks $M^{(1)}, M^{(2)}, \ldots, M^{(N)}$.

The message blocks are processed one at a time: Beginning with a fixed initial hash value $H^{(0)}$,sequentially compute

$$H^{(i)} = H^{(i-1)} + C_{M^{(i)}}\left(H^{(i-1)}\right),$$

where C is the SHA-256 compression function and + means word-wise mod $2^{64}$ addition. H(N) is the hash of M.

## 1.1 Description of SHA-512

The SHA-256 compression function operates on a 512-bit message block and a 256-bit intermediate hash value. It is essentially a 256-bit block cipher algorithm which encrypts the intermediate hash value using the message block as key. Hence there are two main components to describe: (1) the SHA-256 compression function, and (2) the SHA-256 message schedule. We will use the following notation:

| | |
|---|---|
| $\oplus$ | bitwise XOR |
| $\wedge$ | bitwise AND |
| $\vee$ | bitwise OR |
| $\neg$ | bitwise complement |
| $+$ | mod $2^{64}$ addition |
| $R^n$ | right shift by n bits |
| $S^n$ | right rotation by n bits |

All of these operators act on 32-bit words.

The initial hash value $H^{(0)}$ is the following sequence of 32-bit words (which are obtained by taking the fractional parts of the square roots of the first eight primes):

$$H_1^{(0)} = 6a09e667$$

$$H_2^{(0)} = bb67ae85$$

$$H_3^{(0)} = 3c6ef372$$

$$H_4^{(0)} = a54ff53a$$

$$H_5^{(0)} = 510e527f$$

$$H_6^{(0)} = 9b05688c$$

$$H_7^{(0)} = 1f83d9ab$$

$$H_8^{(0)} = 5be0cd19$$

## Pre-processing

Computation of the hash of a message begins by preparing the message:

1. Pad the message in the usual way: Suppose the length of the message M, in bits, is $l$. Append the bit "1" to the end of the message, and then $k$ zero bits, where $k$ is the smallest non-negative solution to the equation $l + 1 + k \equiv 448 \; mod \; 512$.To this append the 64-bit block which is equal to the number $l$ written in binary. For example, the (8-bit ASCII) message "abc" has length $8 \cdot 3 = 24$ so it is padded with a one, then $448 - (24 + 1) = 423$ zero bits, and then its length to become the 512-bit padded message
$$0110001\ 01100010\ 01100011\ 1\ \underbrace{00\ldots0}_{423}\ \underbrace{00\ldots011000}_{64}.$$
The length of the padded message should now be a multiple of 512bits.

2. Parse the message into $N$ 512-bit blocks $M^{(1)}, M^{(2)}, \ldots, M^{(N)}$. The first 32bits of message block $i$ are denoted $M_0^{(i)}$, the next 32bits are $M_1^{(i)}$, and soon up to $M_{15}^{(i)}$.We use the big-endian convention throughout, so within each 32-bit word, the left-most bit is stored in the most significant bit position.

### Main loop

The hash computation proceeds as follows:

For $i = 1$ to $N$ (N = number of blocks in the padded message)

{

/* Initialize registers $a;\ b;\ c;\ d;\ e;\ f;\ g;\ h$ with the $(i-1)^{st}$ intermediate hash value (= the initial hash value when $i = 1$) */

$a \leftarrow H_1^{(i-1)}$

$b \leftarrow H_2^{(i-1)}$

$\vdots$

$h \leftarrow H_8^{(i-1)}$

/* Apply the SHA-256 compression function to update registers $a; b; \ldots; h$ */

For $j = 0$ to $63$
{

Compute $Ch(e,f,g)$, $Maj(a,b,c)$, $\sum_0(a)$, $\sum_1(e)$, and $Wj$ (see Definitions below)

$T_1 \leftarrow h + \sum_1(e) + Ch(e,f,g) + K_j + W_j$
$T_2 \leftarrow \sum_0(a) + Maj(a,b,c)$
$h \leftarrow g$
$g \leftarrow f$
$f \leftarrow e$
$e \leftarrow d + T_1$
$d \leftarrow c$
$c \leftarrow b$
$b \leftarrow a$
$a \leftarrow T_1 + T_2$
}

// Compute the $i^{th}$ intermediate hash value $H^{(i)}$

$H_1^{(i)} \leftarrow a + H_1^{(i-1)}$
$H_2^{(i)} \leftarrow a + H_2^{(i-1)}$
$\vdots$
$H_8^{(i)} \leftarrow a + H_8^{(i-1)}$

}
$H^{(N)} = (H_1^{(N)}, H_2^{(N)}, \ldots, H_8^{(N)})$ is the hash of $M$.

## Definitions

Six logical functions are used in SHA-256. Each of these functions operates on 32-bit words and produces a 32-bit word as output. Each function is defined as follows:

$Ch(x,y,z) = (x \wedge \mathrm{y}) \oplus (\neg x \wedge \mathrm{z})$
$Maj(x,y,z) = (x \wedge \mathrm{y}) \oplus (x \wedge \mathrm{z}) \oplus (y \wedge \mathrm{z})$
$\sum_0(x) = S^2(x) \oplus S^{13}(x) \oplus S^{22}(x)$
$\sum_1(x) = S^2(x) \oplus S^{13}(x) \oplus S^{22}(x)$
$\sigma_0(x) = S^7(x) \oplus S^{18}(x) \oplus R^3(x)$
$\sigma_1(x) = S^{17}(x) \oplus S^{19}(x) \oplus R^{10}(x)$

Expanded message blocks $W_0, W_1, \ldots W_{63}$ are computed as follows via the SHA-256 message schedule:

$W_j = M_j^{(i)}$ for $j = 0, 1, \ldots, 15$, and
For $j = 16$ to $63$
{

$$W_j \leftarrow \sigma_1(W_{j-2}) + W_{j-7} + \sigma_0(W_{j-15}) + W_{j-16}$$

}

A sequence of constant words, $K_0, K_1, \ldots K_{63}$ , is used in SHA-256. In hex, these are given by

```
428a2f98 71374491 b5c0fbcf e9b5dba5 3956c25b 59f111f1 923f82a4 ab1c5ed5
d807aa98 12835b01 243185be 550c7dc3 72be5d74 80deb1fe 9bdc06a7 c19bf174
e49b69c1 efbe4786 0fc19dc6 240ca1cc 2de92c6f 4a7484aa 5cb0a9dc 76f988da
983e5152 a831c66d b00327c8 bf597fc7 c6e00bf3 d5a79147 06ca6351 14292967
27b70a85 2e1b2138 4d2c6dfc 53380d13 650a7354 766a0abb 81c2c92e 92722c85
a2bfe8a1 a81a664b c24b8b70 c76c51a3 d192e819 d6990624 f40e3585 106aa070
19a4c116 1e376c08 2748774c 34b0bcb5 391c0cb3 4ed8aa4a 5b9cca4f 682e6ff3
748f82ee 78a5636f 84c87814 8cc70208 90befffa a4506ceb bef9a3f7 c67178f2
```

These are the first thirty-two bits of the fractional parts of the cube roots of the first sixty-four primes.

## 1.3 Diagrams

The SHA-256 compression function is pictured below:

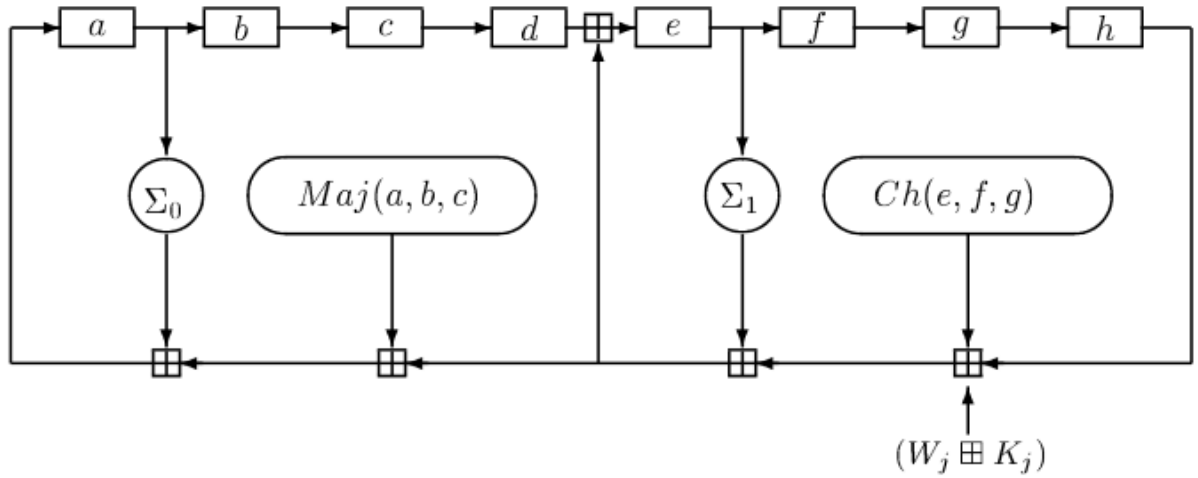



**Figure 1**: $j^{\text{th}}$ internal step of the SHA-256 compression function $C$

Where the symbol $\boxplus$ denotes mod $2^{32}$ addition.

The message schedule can be drawn as follows:

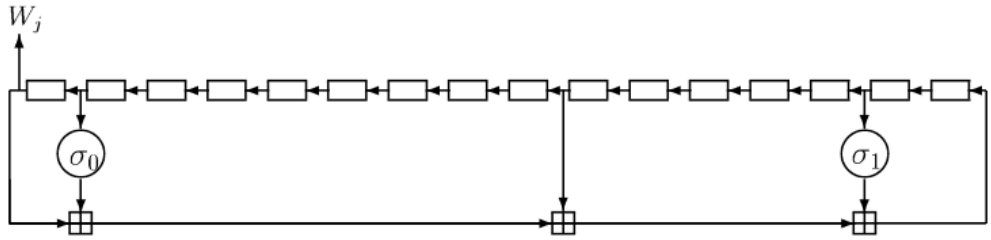



**Figure 2**: SHA-256 message schedule

The registers here are loaded with $W_0, W_1, \ldots W_{15}$.

| Component | Links |
|---|---|
| ATECC508A<br>ATECC508C | https://www.geeksforgeeks.org/implementation-diffie-hellman-algorithm/ |

Elliptic Curve Diffie-Helmman key agreement

## Background

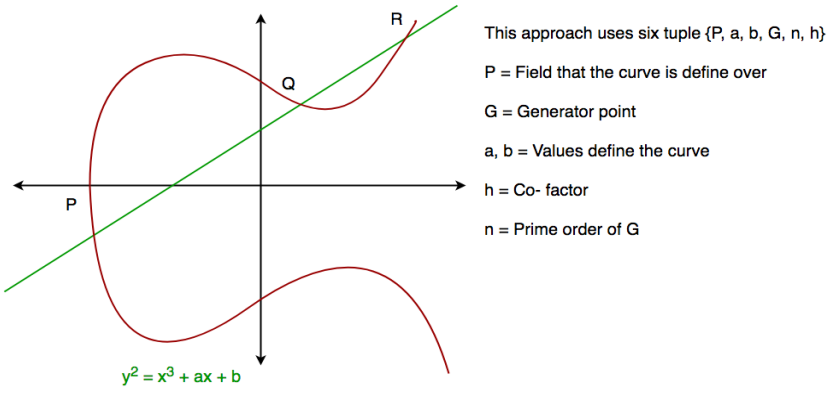
**Elliptic Curve Cryptography (ECC)** is an approach to public-key cryptography, based on the algebraic structure of elliptic curves over finite fields. ECC requires a smaller key as compared to non-ECC cryptography to provide equivalent security (a 256-bit ECC security have an equivalent security attained by 3072-bit RSA cryptography).

For a better understanding of Elliptic Curve Cryptography, it is very important to understand the basics of Elliptic Curve. An elliptic curve is a planar algebraic curve defined by an equation of the form

$$y^2 = x^3 + ax + b$$

where 'a' is the co-efficient of x and 'b' is the constant of the equation

The curve is non-singular; that is its graph has no cusps or self-intersections (when the characteristic of the
co-efficient field is equal to 2 or 3).
In general, an elliptic curve looks like as shown below. Elliptic curves could intersect atmost 3 points when a straight line is drawn intersecting the curve. As we can see that elliptic curve is symmetric about the x-axis, this property plays a key role in the algorithm.




## Diffie-Hellman algorithm

The Diffie-Hellman algorithm is being used to establish a shared secret that can be used for secret
communications while exchanging data over a public network using the elliptic curve to generate points and get the secret key using the parameters.

- For the sake of simplicity and practical implementation of the algorithm, we will consider only 4 variables one prime P and G (a primitive root of P) and two private values a and b.
- P and G are both publicly available numbers. Users (say Alice and Bob) pick private values a and b and they generate a key and exchange it publicly, the opposite person received the key and from that generates a secret key after which they have the same secret key to encrypt.

**Step by Step Explanation**

| Alice | Bob |
|---|---|
| Public Keys available = P, G | Public Keys available = P, G |
| Private Key Selected = a | Private Key Selected = b |
| Key generated = $x = G^a \ mod \ P$ | Key generated = y= $G^b \ mod \ P$ |
| Exchange of generated keys takes place | |
| Key received = $y$ | key received = $x$ |
| Generated Secret Key = $k_a = y^a \ mod \ P$ | Generated Secret Key = $k_b = x^b \ mod \ P$ |
| Algebraically it can be shown that $k_a = k_b$ | |
| Users now have a symmetric secret key to encrypt | |

| Component | Links |
|---|---|
| ATAES132A | https://www.korelstar.de/informatik/aes.html<br>https://en.wikipedia.org/wiki/Advanced_Encryption_Standard#Description_of_the_ciphers |

Advanced Encryption Standard is a part of the Rijndael block cipher developed by Vincent Rijmen and Joan Daemen.

The cypher works as a substitution-permutation network. AES has a fixed block size of 128 bits and key lengths of 128, 192 or 256.

Algorithm

The 128bit message block gets represented as a two-dimensional array:

$$\begin{bmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{bmatrix}$$

Depending on the Key size a different number of rounds get done.

| Rounds | Key length |
|---|---|
| 10 | 128-bit |
| 12 | 192-bit |
| 14 | 256-bit |

Processing steps (1:1 von Wiki)

1. KeyExpansion-round keys re derived from the cipher key using Rijndael's key schedule. AES requires a separate 128-bit round key block for each round plus one more.
2. Initial round key addition:
    a. AddRoundKey—each byte of the state is combined with a block of the round key using bitwise xor.
3. n - 1 rounds
    a. SubBytes—a non-linear substitution step where each byte is replaced with another according to a lookup table.
    b. ShiftRows—a transposition step where the last three rows of the state are shifted cyclically a certain number of steps.
    c. MixColumns—a linear mixing operation which operates on the columns of the state, combining the four bytes in each column.
    d. AddRoundKey
4. Final round
    a. SubBytes
    b. ShiftRows
    c. AddRoundKey

## SubBytes

In the SubBytes step, each byte $a_{i,j}$ in the *state* array is replaced with a SubByte $S(a_{i,j})$ using an 8-bit substitution box. This operation provides the non-linearity in the cipher. The S-box used is derived from the multiplicative inverse over GF($2^8$), known

to have good non-linearity properties. To avoid attacks based on simple algebraic properties, the S-box is constructed by combining the inverse function with an invertible affine transformation. The S-box is also chosen to avoid any fixed points (and so is a derangement), i.e., $S(a_{i,j}) \neq a_{i,j}$, and also any opposite fixed points, i.e., $S(a_{i,j}) \oplus b_{i,j} \neq FF_{16}$ . While performing the decryption, the InvSubBytes step (the inverse of SubBytes) is used, which requires first taking the inverse of the affine transformation and then finding the multiplicative inverse.

$$S(a_{i,j}) = b_{i,j}$$

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \xrightarrow{SubBytes} \begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix}$$

## ShiftRows

The ShiftRows step operates on the rows of the state; it cyclically shifts the bytes in each row by a certain offset. For AES, the first row is left unchanged. Each byte of the second row is shifted one to the left. Similarly, the third and fourth rows are shifted by offsets of two and three respectively. In this way, each column of the output state of the ShiftRows step is composed of bytes from each column of the input state. The importance of this step is to avoid the columns being encrypted independently, in which case AES degenerates into four independent block ciphers.

$$\begin{matrix} no\ change \\ shift1 \\ shift\ 2 \\ shift\ 3 \end{matrix} \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \xrightarrow{ShiftRows} \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,2} & a_{1,2} & a_{1,3} & a_{1,0} \\ a_{2,2} & a_{2,3} & a_{2,0} & a_{2,1} \\ a_{3,3} & a_{3,0} & a_{3,1} & a_{3,2} \end{bmatrix}$$

## MixColumns

In the MixColumns step, the four bytes of each column of the state are combined using an invertible linear transformation. The MixColumns function takes four bytes as input and outputs four bytes, where each input byte affects all four output bytes. Together with ShiftRows, MixColumns provides diffusion in the cipher. During this operation, each column is transformed using a fixed matrix (matrix left-multiplied by column gives new value of column in the state):

$$\begin{bmatrix} b_{0,j} \\ b_{1,j} \\ b_{2,j} \\ b_{3,j} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} a_{0,j} \\ a_{1,j} \\ a_{2,j} \\ a_{3,j} \end{bmatrix} \quad 0 \leq j \leq 3$$

Matrix multiplication is composed of multiplication and addition of the entries. Entries are 8-bit bytes treated as coefficients of polynomial of order . Addition is simply XOR. Multiplication is modulo irreducible polynomial $x^8 + x^4 + x^3 + x + 1$. If processed bit by bit, then, after shifting, a conditional XOR with $1B_{16}$ should be performed if the shifted value is larger than $FF_{16}$ (overflow must be corrected by subtraction of generating polynomial). These are special cases of the usual multiplication in GF($2^8$).

In more general sense, each column is treated as a polynomial over GF($2^8$) and is then multiplied modulo $01_{16} \cdot z^4 + 01_{16}$ with a fixed polynomial $c(z) = 03_{16} \cdot z^3 + 01_{16} \cdot z^2 + 01_{16} \cdot z + 02_{16}$. The coefficients are displayed in their hexadecimal equivalent of the binary representation of bit polynomials from . The MixColumns step can also be viewed as a multiplication by the shown particular MDS matrix in the finite field GF($2^8$).

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \xrightarrow{MixColumns} \begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix}$$

## AddRoundKey

In the AddRoundKey step, the subkey is combined with the state. For each round, a subkey is derived from the main key using Rijndael's key schedule; each subkey is the same size as the state. The subkey is added by combining each byte of the state with the corresponding byte of the subkey using bitwise XOR.

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \begin{bmatrix} k_{0,0} & k_{0,1} & k_{0,2} & k_{0,3} \\ k_{1,0} & k_{1,1} & k_{1,2} & k_{1,3} \\ k_{2,0} & k_{2,1} & k_{2,2} & k_{2,3} \\ k_{3,0} & k_{3,1} & k_{3,2} & k_{3,3} \end{bmatrix} \xrightarrow{\text{XOR}} \begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix}$$