SHA-256 Cryptographic Hash Algorithm

A **cryptographic hash** (sometimes called 'digest') is a kind of 'signature' for a text or a data file. SHA-256 generates an almost-unique 256-bit (32-byte) signature for a text. See below for the source code.

Enter any message to check its SHA-256 hash

Message: abc

Hash: ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad 0.500ms

Note SHA-256 hash of 'abc' should be: ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad

A hash is not 'encryption' – it cannot be decrypted back to the original text (it is a 'one-way' cryptographic function, and is a fixed size for any size of source text). This makes it suitable when it is appropriate to compare 'hashed' versions of texts, as opposed to decrypting the text to obtain the original version.

Such applications include hash tables, integrity verification, challenge handshake authentication, digital signatures, etc.

- '*challenge handshake authentication*' (or 'challenge hash authentication') avoids transmissing passwords in 'clear' – a client can send the hash of a password over the internet for validation by a server without risk of the original password being intercepted
- *anti-tamper* – link a hash of a message to the original, and the recipient can re-hash the message and compare it to the supplied hash: if they match, the message is unchanged; this can also be used to confirm no data-loss in transmission
- *digital signatures* are rather more involved, but in essence, you can sign the hash of a document by encrypting it with your private key, producing a digital signature for the document. Anyone else can then check that you authenticated the text by decrypting the signature with your public key to obtain the original hash again, and comparing it with their hash of the text.

Note that hash functions are not appropriate for storing encrypted passwords, as they are designed to be fast to compute, and hence would be candidates for brute-force attacks. Key derivation functions such as bcrypt or scrypt are designed to be slow to compute, and are more appropriate for password storage (npm has bcrypt and scrypt libraries, and PHP has a bcrypt implementation with password_hash).

SHA-256 is one of the successor hash functions to SHA-1 (collectively referred to as SHA-2), and is one of the strongest hash functions available. SHA-256 is not much more complex to code than SHA-1, and has not yet been compromised in any way. The 256-bit key makes it a good partner-function for AES. It is defined in the NIST (National Institute of Standards and Technology) standard 'FIPS 180-4'. NIST also provide a number of test vectors to verify correctness of implementation. There is a good description at Wikipedia.

In this **JavaScript implementation**, I have tried to make the script as clear and concise as possible, and equally as close as possible to the NIST specification, to make the operation of the script readily understandable.

This script is oriented toward hashing text messages rather than binary data. The standard considers hashing byte-stream (or bit-stream) messages only. Text which contains (multi-byte) characters outside ISO 8859-1 (i.e. accented characters outside Latin-1 or non-European character sets – anything with Unicode code-point above U+FF), can't be encoded 4-per-word, so the script defaults to encoding the text as UTF-8 before hashing it.

Notes on the implementation of the preprocessing stage:

- FIPS 180-4 specifies that the message has a '1' bit appended, and is then padded to a whole number of 512-bit

blocks, including the message length (in bits) in the final 64 bits of the last block
- Since we have a byte-stream rather than a bit-stream, adding a byte '10000000' (0x80) appends the required bit "1".
- To convert the message to 512-bit blocks, I calculate the number of blocks required, N, then for each of these I create a 16-integer (i.e. 512-bit) array. For each if these integers, I take four bytes from the message (using charCodeAt), and left-shift them by the appropriate amount to pack them into the 32-bit integer.
- The charCodeAt() method returns NaN for out-of-bounds, but the '|' operator converts this to zero, so the 0-padding is done implicitly on conversion into blocks.
- Then the length of the message (in bits) needs to be appended in the last 64 bits, that is the last two integers of the final block. In principle, this could be done by

  ```
  M[N-1][14] = ((msg.length-1)*8) >>> 32;
  M[N-1][15] = ((msg.length-1)*8) & 0xffffffff;
  ```

  However, JavaScript bit-ops convert their arguments to 32-bits, so n >>> 32 would give 0. Hence I use arithmetic operators instead: for the most-significant 32-bit number, I divide the (original) length by 2^32, and use floor() convert the result to an integer.

Note that what is returned is the textual hexadecimal representation of the binary hash. This can be useful for instance for storing hashed passwords, but if you want to use the hash as a key to an encryption routine, for example, you will want to use the binary value not this textual representation.

Using Chrome on a low-to-middling Core i5 PC, in timing tests this script will hash a short message in around 0.03 – 0.06 ms; longer messages will be hashed at a speed of around 2 – 3 MB/sec.

If you are interested in the simpler SHA-1, I have a JavaScript implementation of SHA-1. I have also implemented SHA-512 and SHA-3 / Keccak.

If you are interested in encryption rather than a cryptographic hash algorithm, look at my JavaScript implementation of TEA (Tiny Encryption Algorithm) or JavaScript implementation of AES.

Note that these scripts are intended to assist in studying the algorithms, not for production use. For production use, I would recommend the *Web Cryptography API* for the browser (see *example*), or the *crypto* library in Node.js. For password hashing, I have a WebCrypto *example using PBKDF2*.

See below for the source code of the JavaScript implementation, also available on GitHub. §ection numbers relate the code back to sections in the standard. Note I use Greek letters in the 'logical functions', as presented in the spec (if you encounter any problems, ensure your `<head>` includes `<meta charset="utf-8">`).

With its untyped C-style syntax, JavaScript reads remarkably close to pseudo-code: exposing the algorithms with a minimum of syntactic distractions. These functions should be simple to translate into other languages if required, though can also be used as-is in browsers and Node.js.

If you would like to show your appreciation and support continued development of these scripts, I would most gratefully accept donations.

If you have any queries or find any problems, contact me at scripts-enc@movable-type.co.uk.

```
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
/* SHA-256 (FIPS 180-4) implementation in JavaScript                  (c) Chris Veness 2002-2019  */
/*                                                                                 MIT Licence  */
/* www.movable-type.co.uk/scripts/sha256.html                                                   */
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
```

```
/**
 * SHA-256 hash function reference implementation.
 *
 * This is an annotated direct implementation of FIPS 180-4, without any optimisations. It is
 * intended to aid understanding of the algorithm rather than for production use.
 *
 * While it could be used where performance is not critical, I would recommend using the 'Web
 * Cryptography API' (developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/digest) for the browser,
 * or the 'crypto' library (nodejs.org/api/crypto.html#crypto_class_hash) in Node.js.
 *
 * See csrc.nist.gov/groups/ST/toolkit/secure_hashing.html
 *     csrc.nist.gov/groups/ST/toolkit/examples.html
 */
class Sha256 {

    /**
     * Generates SHA-256 hash of string.
     *
     * @param   {string} msg - (Unicode) string to be hashed.
     * @param   {Object} [options]
     * @param   {string} [options.msgFormat=string] - Message format: 'string' for JavaScript string
     *   (gets converted to UTF-8 for hashing); 'hex-bytes' for string of hex bytes ('616263' ≡ 'abc') .
     * @param   {string} [options.outFormat=hex] - Output format: 'hex' for string of contiguous
     *   hex bytes; 'hex-w' for grouping hex bytes into groups of (4 byte / 8 character) words.
     * @returns {string} Hash of msg as hex character string.
     *
     * @example
     *   import Sha256 from './sha256.js';
     *   const hash = Sha256.hash('abc'); // 'ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad'
     */
    static hash(msg, options) {
        const defaults = { msgFormat: 'string', outFormat: 'hex' };
        const opt = Object.assign(defaults, options);

        // note use throughout this routine of 'n >>> 0' to coerce Number 'n' to unsigned 32-bit integer

        switch (opt.msgFormat) {
            default: // default is to convert string to UTF-8, as SHA only deals with byte-streams
            case 'string':   msg = utf8Encode(msg);       break;
            case 'hex-bytes':msg = hexBytesToString(msg); break; // mostly for running tests
        }

        // constants [§4.2.2]
        const 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 ];

        // initial hash value [§5.3.3]
        const H = [
            0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a, 0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19 ];

        // PREPROCESSING [§6.2.1]

        msg += String.fromCharCode(0x80);  // add trailing '1' bit (+ 0's padding) to string [§5.1.1]

        // convert string msg into 512-bit blocks (array of 16 32-bit integers) [§5.2.1]
        const l = msg.length/4 + 2; // length (in 32-bit integers) of msg + '1' + appended length
        const N = Math.ceil(l/16);  // number of 16-integer (512-bit) blocks required to hold 'l' ints
        const M = new Array(N);     // message M is N×16 array of 32-bit integers

        for (let i=0; i<N; i++) {
            M[i] = new Array(16);
            for (let j=0; j<16; j++) { // encode 4 chars per integer (64 per block), big-endian encoding
                M[i][j] = (msg.charCodeAt(i*64+j*4+0)<<24) | (msg.charCodeAt(i*64+j*4+1)<<16)
                        | (msg.charCodeAt(i*64+j*4+2)<< 8) | (msg.charCodeAt(i*64+j*4+3)<< 0);
            } // note running off the end of msg is ok 'cos bitwise ops on NaN return 0
        }
        // add length (in bits) into final pair of 32-bit integers (big-endian) [§5.1.1]
```

```
        // note: most significant word would be (len-1)*8 >>> 32, but since JS converts
        // bitwise-op args to 32 bits, we need to simulate this by arithmetic operators
        const lenHi = ((msg.length-1)*8) / Math.pow(2, 32);
        const lenLo = ((msg.length-1)*8) >>> 0;
        M[N-1][14] = Math.floor(lenHi);
        M[N-1][15] = lenLo;


        // HASH COMPUTATION [§6.2.2]

        for (let i=0; i<N; i++) {
            const W = new Array(64);

            // 1 - prepare message schedule 'W'
            for (let t=0;  t<16; t++) W[t] = M[i][t];
            for (let t=16; t<64; t++) {
                W[t] = (Sha256.σ1(W[t-2]) + W[t-7] + Sha256.σ0(W[t-15]) + W[t-16]) >>> 0;
            }

            // 2 - initialise working variables a, b, c, d, e, f, g, h with previous hash value
            let a = H[0], b = H[1], c = H[2], d = H[3], e = H[4], f = H[5], g = H[6], h = H[7];

            // 3 - main loop (note '>>> 0' for 'addition modulo 2^32')
            for (let t=0; t<64; t++) {
                const T1 = h + Sha256.Σ1(e) + Sha256.Ch(e, f, g) + K[t] + W[t];
                const T2 =     Sha256.Σ0(a) + Sha256.Maj(a, b, c);
                h = g;
                g = f;
                f = e;
                e = (d + T1) >>> 0;
                d = c;
                c = b;
                b = a;
                a = (T1 + T2) >>> 0;
            }

            // 4 - compute the new intermediate hash value (note '>>> 0' for 'addition modulo 2^32')
            H[0] = (H[0]+a) >>> 0;
            H[1] = (H[1]+b) >>> 0;
            H[2] = (H[2]+c) >>> 0;
            H[3] = (H[3]+d) >>> 0;
            H[4] = (H[4]+e) >>> 0;
            H[5] = (H[5]+f) >>> 0;
            H[6] = (H[6]+g) >>> 0;
            H[7] = (H[7]+h) >>> 0;
        }

        // convert H0..H7 to hex strings (with leading zeros)
        for (let h=0; h<H.length; h++) H[h] = ('00000000'+H[h].toString(16)).slice(-8);

        // concatenate H0..H7, with separator if required
        const separator = opt.outFormat=='hex-w' ? ' ' : '';

        return H.join(separator);

        /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  */

        function utf8Encode(str) {
            try {
                return new TextEncoder().encode(str, 'utf-8').reduce((prev, curr) => prev + String.fromCharCode(curr), '');
            } catch (e) { // no TextEncoder available?
                return unescape(encodeURIComponent(str)); // monsur.hossa.in/2012/07/20/utf-8-in-javascript.html
            }
        }

        function hexBytesToString(hexStr) { // convert string of hex numbers to a string of chars (eg '616263' -> 'abc').
            const str = hexStr.replace(' ', ''); // allow space-separated groups
            return str=='' ? '' : str.match(/.{2}/g).map(byte => String.fromCharCode(parseInt(byte, 16))).join('');
        }
    }



    /**
     * Rotates right (circular right shift) value x by n positions [§3.2.4].
     * @private
```

```
     */
    static ROTR(n, x) {
        return (x >>> n) | (x << (32-n));
    }


    /**
     * Logical functions [§4.1.2].
     * @private
     */
    static Σ0(x) { return Sha256.ROTR(2,  x) ^ Sha256.ROTR(13, x) ^ Sha256.ROTR(22, x); }
    static Σ1(x) { return Sha256.ROTR(6,  x) ^ Sha256.ROTR(11, x) ^ Sha256.ROTR(25, x); }
    static σ0(x) { return Sha256.ROTR(7,  x) ^ Sha256.ROTR(18, x) ^ (x>>>3);  }
    static σ1(x) { return Sha256.ROTR(17, x) ^ Sha256.ROTR(19, x) ^ (x>>>10); }
    static Ch(x, y, z)  { return (x & y) ^ (~x & z); }          // 'choice'
    static Maj(x, y, z) { return (x & y) ^ (x & z) ^ (y & z); } // 'majority'

}


/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  */

export default Sha256;
```