# libft_ssl

Cryptographic Hash Functions

lohhiiccc

## Contents

# 1 Preliminaries

This section defines the terminology used throughout the document. The concepts introduced here are general to cryptographic hash functions and apply to all algorithms described in subsequent sections.

A **bit** is the smallest unit of information, taking a value of either 0 or 1. A **byte** is a group of eight bits, and is the standard unit of data storage and transmission. A **word** is a fixed-size integer used internally by a hash algorithm — MD5 and SHA-256 operate on 32-bit words, while Whirlpool operates on 64-bit words.

**Endianness** refers to the byte order used when storing a multi-byte integer in memory. In **little-endian** order, the least significant byte is stored first; in **big-endian** order, the most significant byte is stored first. This distinction matters when serializing the internal state to produce the final digest — MD5 uses little-endian, while SHA-256 and Whirlpool use big-endian.

$$\text{Value } 0xD41D8CD9 : \begin{cases} \text{Big-endian:} & \text{d4 | 1d | 8c | d9} \\ \text{Little-endian:} & \text{d9 | 8c | 1d | d4} \end{cases}$$

A **message** is the arbitrary-length input fed to a hash function. The **digest** is the fixed-size output it produces. A hash function is said to be **one-way** if it is computationally infeasible to recover any input that produces a given digest. A **collision** occurs when two distinct messages produce the same digest; a hash function is considered broken when collisions can be found efficiently.

Hash functions process their input in fixed-size chunks called **blocks**. Since the message length is rarely a multiple of the block size, **padding** is appended to the last block to bring it to the required length. The **state** is a set of words initialized to fixed constants and updated after each block; it accumulates the result of the computation and is serialized into the digest at the end. The **compression function** is the core transformation applied to each block — it takes the current state and one block of data, and produces a new state.

The **Miyaguchi-Preneel** construction is a way to build a compression function from a block cipher $E$. Given a current state $H$ and a message block $M$, it produces a new state as:

$$H \leftarrow E(H,\ M) \oplus M \oplus H$$

where $E(H, M)$ denotes the encryption of $M$ using $H$ as the key. The XOR with both $M$ and $H$ ensures that the output cannot be trivially inverted even if $E$ is known.

The **wide-pipe** construction is a variant of Merkle-Damgård where the internal state is wider than the final digest. This makes collision attacks harder: an attacker targeting the output must first find a collision in the larger internal state, which requires significantly more work than attacking the digest directly.

# 2 Introduction

`libft_ssl` is a C library implementing cryptographic hash functions from scratch. A cryptographic hash function maps an arbitrary-length input to a fixed-size digest. This operation is deterministic and one-way: it is computationally infeasible to recover the original input from its digest.

The library currently implements the following algorithms:

- **MD5** - produces a 128-bit digest.
- **SHA-256** - produces a 256-bit digest.
- **Whirlpool** - produces a 512-bit digest.

These functions are commonly used for data integrity verification, digital signatures, and **M**essage **A**uthentication **C**ode**s** (MACs).

# 3 Generic Digest Interface

All hash algorithms in `libft_ssl` are exposed through a single, uniform interface built around the `struct digest_algo` type. This structure holds the algorithm's metadata (its name, digest size and block size) along with three function pointers: `init`, `update` and `final`. This design allows any algorithm to be driven through the same calling convention without the caller needing to know which one is in use. The associated context is held in a `union digest_ctx`, which overlays the per-algorithm state structures so that a single allocation covers all supported algorithms.

`struct digest_algo` describes a hash algorithm as a set of metadata and three function pointers. The `name` field identifies the algorithm. The `digest_size` and `block_size` fields express its output length and internal block size in bytes. The three function pointers `init`, `update` and `final` define the algorithm's lifecycle: `init` sets the context to its initial state, `update` feeds an arbitrary amount of data into it, and `final` produces the digest and resets the context. All three operate on a `void *` context pointer, which allows the interface to remain algorithm-agnostic.

The `union digest_ctx` type provides a single allocation large enough to hold the context of any supported algorithm. Because only one algorithm is active at a time, overlaying the per-algorithm structures in a union avoids the overhead of a separate heap allocation while keeping the calling code uniform. The active member is always the one matching the `struct digest_algo` being used.

Each supported algorithm is registered in `digest_algos.h` through an X-macro list. This file defines a single macro `DIGEST_ALGOS(X)` that expands `X` once per algorithm, passing its name, digest size and block size. Consuming this list with a different definition of `X` generates the corresponding code or data without repetition — the global `struct digest_algo` instances in `libft_ssl.c` are produced this way. Adding a new algorithm to the library reduces to adding one line to this list.

All three algorithms follow the Merkle-Damgård construction. The message is split into fixed-size blocks and processed sequentially. After each block, the compressed output is combined with the previous state to produce the new state — this chaining ensures that the final digest depends on every bit of the input. The exact combination operation is algorithm-specific: MD5 and SHA-256 use an additive feedforward, while Whirlpool uses the Miyaguchi-Preneel scheme.

# 4 MD5

MD5 (Message Digest Algorithm 5) was designed by Ronald Rivest in 1991 as a strengthened replacement for MD4. It produces a 128-bit digest from a message of arbitrary length, processing data in 512-bit blocks. Although MD5 is now considered cryptographically broken (collision attacks have been demonstrated since 2004) it remains widely used for non-security purposes such as checksums and data integrity verification.

MD5 maintains a state of four 32-bit words, conventionally named $A$, $B$, $C$ and $D$, initialized to fixed constants defined in RFC 1321. Each 512-bit block is processed in four rounds of sixteen operations each, for a total of 64 operations per block. Each operation applies one of four non-linear functions to the state words, adds a message word and a precomputed constant derived from the sine function, and rotates the result by a fixed amount.

The state is initialized to the following fixed constants, as specified in RFC 1321:

$$\begin{aligned} A &= \texttt{0x67452301} \\ B &= \texttt{0xefcdab89} \\ C &= \texttt{0x98badcfe} \\ D &= \texttt{0x10325476} \end{aligned}$$

Before processing, the message is padded to a length congruent to 448 bits modulo 512. A single `1` bit is appended first, followed by as many `0` bits as needed. The original message length in bits is then appended as a 64-bit little-endian integer, bringing the total padded length to an exact multiple of 512 bits.

Each of the four rounds uses a distinct non-linear function applied to the state words $B$, $C$ and $D$:

$$\begin{aligned} F(B, C, D) &= (B \wedge C) \vee (\neg B \wedge D) \\ G(B, C, D) &= (B \wedge D) \vee (C \wedge \neg D) \\ H(B, C, D) &= B \oplus C \oplus D \\ I(B, C, D) &= C \oplus (B \vee \neg D) \end{aligned}$$

The message word index used at step $i$ is not sequential: each round applies a distinct selector function $k_r$ where $r = \lfloor i/16 \rfloor$:

$$\begin{aligned} k_0(i) &= i \bmod 16 \\ k_1(i) &= (5i + 1) \bmod 16 \\ k_2(i) &= (3i + 5) \bmod 16 \\ k_3(i) &= 7i \bmod 16 \end{aligned}$$

At each step $i$ (with $0 \leq i < 64$), one of the four functions is selected according to the current round, and the state is updated as follows:

$$A \leftarrow B + \big((A + \phi(B, C, D) + M[k] + T[i]) \lll s[i]\big)$$

where $\phi$ is the auxiliary function for the current round, $M[k]$ is a 32-bit word of the current block, $T[i]$ is a precomputed constant, $s[i]$ is the rotation amount, and $\lll$ denotes a left rotation. After this operation, the state words are cycled: $(A, B, C, D) \leftarrow (D, A, B, C)$.

The rotation amounts $s[i]$ are constant per round and repeat every four steps:

$$\begin{aligned}
&\text{Round } 0 : 7,\ 12,\ 17,\ 22 \\
&\text{Round } 1 : 5,\ 9,\ 14,\ 20 \\
&\text{Round } 2 : 4,\ 11,\ 16,\ 23 \\
&\text{Round } 3 : 6,\ 10,\ 15,\ 21
\end{aligned}$$

The 64 constants $T[i]$ are derived from the sine function:

$$\forall i \in \mathbb{N},\ 0 \leq i < 64, T_i = \left\lfloor 2^{32} \left| \sin(i + 1) \right| \right\rfloor$$

After each block is processed, the compressed state is added word-by-word to the state before compression:

$$(A, B, C, D) \leftarrow (A + A_0,\ B + B_0,\ C + C_0,\ D + D_0)$$

where $A_0$, $B_0$, $C_0$, $D_0$ denote the state at the beginning of the block. After all blocks have been processed, the four state words are serialized in little-endian order to produce the 128-bit digest.

# 5 SHA-256

SHA-256 is part of the SHA-2 family of cryptographic hash functions, designed by the NSA and first published by NIST in 2001. It produces a 256-bit digest from a message of arbitrary length, processing data in 512-bit blocks. Unlike MD5, SHA-256 has no known practical collision attacks and remains widely used in security-critical applications such as TLS certificates and Bitcoin's proof-of-work.

SHA-256 maintains a state of eight 32-bit words, initialized to fixed constants derived from the square roots of the first eight prime numbers. Each 512-bit block is processed in 64 rounds. Each round applies a compression step involving two non-linear functions, a message schedule word, and a precomputed constant derived from the cube roots of the first 64 prime numbers.

The padding scheme is identical to MD5: a single `1` bit is appended, followed by `0` bits until the message length is congruent to 448 bits modulo 512, and the original length in bits is appended as a 64-bit integer. The difference is that SHA-256 encodes this length in big-endian order.

Each round uses two non-linear functions applied to the state words:

$$\begin{aligned}\mathrm{Ch}(E, F, G) &= (E \wedge F) \oplus (\neg E \wedge G) \\ \mathrm{Maj}(A, B, C) &= (A \wedge B) \oplus (A \wedge C) \oplus (B \wedge C)\end{aligned}$$

and two rotation-based functions applied to the state words $A$ and $E$:

$$\begin{aligned}\Sigma_0(A) &= (A \ggg 2) \oplus (A \ggg 13) \oplus (A \ggg 22) \\ \Sigma_1(E) &= (E \ggg 6) \oplus (E \ggg 11) \oplus (E \ggg 25)\end{aligned}$$

where $\ggg$ denotes a right rotation.

At each round $i$ (with $0 \leq i < 64$), the state is updated as follows:

$$\begin{aligned}T_1 &= H + \Sigma_1(E) + \mathrm{Ch}(E, F, G) + K[i] + W[i] \\ T_2 &= \Sigma_0(A) + \mathrm{Maj}(A, B, C) \\ H &\leftarrow G, \quad G \leftarrow F, \quad F \leftarrow E, \quad E \leftarrow D + T_1 \\ D &\leftarrow C, \quad C \leftarrow B, \quad B \leftarrow A, \quad A \leftarrow T_1 + T_2\end{aligned}$$

where $K[i]$ is a precomputed constant and $W[i]$ is a word from the message schedule.

$$\forall i \in \mathbb{N},\ 0 \leq i < 64, \quad K[i] = \left\lfloor 2^{32} \times (\sqrt[3]{p_{i+1}} \bmod 1) \right\rfloor$$

where $p_{i+1}$ is the $(i+1)$-th prime number and mod1 denotes the fractional part.

The message schedule extends the 16 words of the current block into 64 words using two additional rotation-based functions:

$$\sigma_0(x) = (x \ggg 7) \oplus (x \ggg 18) \oplus (x \gg 3)$$
$$\sigma_1(x) = (x \ggg 17) \oplus (x \ggg 19) \oplus (x \gg 10)$$

where $\gg$ denotes a logical right shift, and $M[i]$ denotes the $i$-th 32-bit word of the current 512-bit block. The schedule is then defined as:

$$W[i] = \begin{cases} M[i] & i \in \mathbb{N},\ 0 \leq i < 16 \\ \sigma_1(W[i-2]) + W[i-7] + \sigma_0(W[i-15]) + W[i-16] & i \in \mathbb{N},\ 16 \leq i < 64 \end{cases}$$

The state is initialized to fixed constants derived from the square roots of the first eight prime numbers:

$$\forall (X, p) \in \{(A,2),(B,3),(C,5),(D,7),(E,11),(F,13),(G,17),(H,19)\}, X \leftarrow \left\lfloor \{\sqrt{p}\}\ 2^{32} \right\rfloor$$

$$A = \texttt{0x6a09e667}, \quad B = \texttt{0xbb67ae85}, \quad C = \texttt{0x3c6ef372}, \quad D = \texttt{0xa54ff53a}$$
$$E = \texttt{0x510e527f}, \quad F = \texttt{0x9b05688c}, \quad G = \texttt{0x1f83d9ab}, \quad H = \texttt{0x5be0cd19}$$

After each block is processed, the compressed state is added word-by-word to the state before compression:

$$\forall X \in \{A, B, C, D, E, F, G, H\}, \quad X \leftarrow X + X_0$$

where $X_0$ denote the state at the beginning of the block. After all blocks have been processed, the eight state words are serialized in big-endian order to produce the 256-bit digest.

# 6 Whirlpool

Whirlpool is a cryptographic hash function designed by Vincent Rijmen and Paulo Barreto, first published in 2000 and standardized by ISO/IEC in 2004. It produces a 512-bit digest from a message of arbitrary length, processing data in 512-bit blocks. Its internal structure is inspired by the wide-pipe Miyaguchi-Preneel construction and shares design principles with AES, using a substitution-permutation network over an $8 \times 8$ matrix of bytes.

Whirlpool maintains a state of eight 64-bit words, forming an $8 \times 8$ matrix of bytes. Each 512-bit block is processed in 10 rounds. Each round applies four successive transformations to the state matrix: a byte substitution, a column shift, a row mixing, and a round key addition.

The padding scheme appends a single `1` bit, followed by `0` bits until the message length is congruent to 256 bits modulo 512. The original message length in bits is then appended as a 256-bit big-endian integer, bringing the total padded length to an exact multiple of 512 bits. Whirlpool uses a 256-bit length field (rather than the 64-bit field of MD5 and SHA-256) to support messages up to $2^{256} - 1$ bits in length.

**Arithmetic in** $\mathrm{GF}(2^8)$**.** A **finite field** (or Galois field) is a finite set in which addition, subtraction, multiplication and division (by any non-zero element) are all well-defined and satisfy the usual algebraic laws. The simplest example is $\mathrm{GF}(2) = \{0, 1\}$, where addition is XOR and multiplication is AND.

$\mathrm{GF}(2^8)$ extends this to 256 elements by representing each element as a polynomial of degree less than 8 with coefficients in $\{0, 1\}$. A byte $b_7 b_6 \cdots b_0$ encodes the polynomial $b_7 x^7 + b_6 x^6 + \cdots + b_0$; for example, `0b10100011` $= x^7 + x^5 + x + 1$.

This construction is analogous to modular arithmetic: just as $\mathbb{Z}/p\mathbb{Z}$ is a field because $p$ is prime, the set of polynomials with binary coefficients forms a field when reduced modulo an *irreducible* polynomial — one that cannot be factored. Without this reduction, multiplying two polynomials could produce a degree greater than 7, stepping outside the 256-element set. Reducing modulo an irreducible polynomial of degree 8 keeps every result within one byte, and guarantees that every non-zero element has a multiplicative inverse.

**Addition** is coefficient-wise addition modulo 2, equivalent to a bitwise XOR (there is no carry: $1 + 1 = 0$):

$$a + b = a \oplus b$$

**Multiplication by** $x$ (called *xtime*) is a left shift by one bit, followed by a conditional reduction: if the original high bit was 1, the degree of the result would reach 8 and must be reduced modulo the irreducible polynomial $p(x) = x^8 + x^4 + x^3 + x^2 + 1$ (`0x11d`), which means XORing with its low byte `0x1d`:

$$\mathrm{xtime}(a) = \begin{cases} a \ll 1 & \text{if } b_7 = 0 \\ (a \ll 1) \oplus \texttt{0x1d} & \text{if } b_7 = 1 \end{cases}$$

**Multiplication by an arbitrary element** decomposes the multiplier into powers of 2, applies xtime repeatedly for each power, then combines the results with XOR. For example, multiplying by `0x05` $= x^2 + 1$:

$$\texttt{0x05} \cdot a = \text{xtime}(\text{xtime}(a)) \oplus a$$

Each round applies the following four transformations in order:

**SubBytes** replaces each byte of the state matrix by its image under the Whirlpool S-box, a fixed 256-entry lookup table defined in the Whirlpool specification.

**ShiftColumns** cyclically shifts each column $j$ of the state matrix upward by $j$ positions, producing a transposition that spreads bytes across rows. Formally, if $a_{i,j}$ denotes the byte at row $i$, column $j$ of the state matrix, ShiftColumns produces:

$$b_{i,j} = a_{i',\ j} \quad \text{where } i' = (i - j) \bmod 8$$

**MixRows** multiplies each row of the state matrix by a fixed circulant MDS matrix over $\mathrm{GF}(2^8)$ with irreducible polynomial $x^8+x^4+x^3+x^2+1$, providing diffusion across the eight bytes of each row. The MDS matrix is fully determined by its first row $(c_0,\ c_1,\ \ldots,\ c_7)$; the entry at row $j$, column $k$ equals $c_{(j-k) \bmod 8}$. Formally, for each row $i$, each output byte $b_{i,j}$ is computed as:

$$b_{i,j} = \bigoplus_{k=0}^{7} c_{(j-k) \bmod 8} \cdot a_{i,k}$$

where $\cdot$ denotes multiplication in $\mathrm{GF}(2^8)$ and $\oplus$ denotes XOR.

**AddRoundKey** XORs the state with the current round key:

$$S \leftarrow S \oplus K[r]$$

The S-box and the MDS matrix coefficients are fixed tables defined in the Whirlpool specification; their values are too large to reproduce here. The round constants $\mathrm{RC}[r]$, $r \in \mathbb{N},\ 1 \le r \le 10$, are however directly derived from the S-box. Each $\mathrm{RC}[r]$ is an 8-word state where only the first word is non-zero:

$$\begin{aligned} \mathrm{RC}[r][0] &= \sum_{k=0}^{7} S[8(r-1)+k] \cdot 2^{8(7-k)} \\ \mathrm{RC}[r][j] &= 0 \quad \forall j \in \mathbb{N},\ 1 \le j \le 7 \end{aligned}$$

Their role is to break symmetry in the key schedule: without them, a symmetric input state would produce symmetric round keys, weakening the internal block transformation.

The round keys $K[r]$, $r \in \mathbb{N},\ 0 \le r \le 10$, are derived from the current hash state. $K[0]$ is set to the state before processing the block. Each subsequent key is obtained by applying the round function to the previous key with a precomputed round constant, where $\mathrm{Round}(S, K)$ denotes the successive application of SubBytes, ShiftColumns, MixRows, and AddRoundKey with key $K$ to state $S$:

$$\begin{aligned} K[0] &= H \\ K[r] &= \mathrm{Round}(K[r-1],\ \mathrm{RC}[r]) \quad r \in \mathbb{N},\ 1 \le r \le 10 \end{aligned}$$

The block $M$ is then encrypted using these keys under a wide-pipe construction. The final state update follows the Miyaguchi-Preneel scheme:

$$H \leftarrow E(H,\ M) \oplus M \oplus H$$

where $E(H, M)$ denotes the encryption of $M$ with key schedule derived from $H$.

The state is initialized to all zeros. After all blocks have been processed, the eight 64-bit state words are serialized in big-endian order to produce the 512-bit digest.

# 7 References


- R. Rivest, *The MD5 Message-Digest Algorithm*, RFC 1321, IETF, 1992.
  https://www.rfc-editor.org/rfc/rfc1321

- Wikipedia, *SHA-2*.
  https://en.wikipedia.org/wiki/SHA-2

- Whirlpool *reference implementations*.
  https://web.archive.org/web/20171129084214/http://www.larc.usp.br/~pbarreto/WhirlpoolPage.html

- Wikipedia, *AES*.
  https://en.wikipedia.org/wiki/Advanced_Encryption_Standard

- National Institute of Standards and Technology, *Secure Hash Standard (SHS)*, FIPS PUB 180-4, 2015.
  https://doi.org/10.6028/NIST.FIPS.180-4

- Wikipedia, *Finite field arithmetic*.
  https://en.wikipedia.org/wiki/Finite_field_arithmetic

- Wikipedia, *Rijndael MixColumns*.
  https://en.wikipedia.org/wiki/Rijndael_MixColumns

- Wikipedia, *Bitwise operation*.
  https://en.wikipedia.org/wiki/Bitwise_operation

- Wikipedia, *Miyaguchi Preneel*.
  https://en.wikipedia.org/wiki/One-way_compression_function#Miyaguchi.E2.80.93Preneel