**Solidity Cryptographic Functions**

**Table of Contents**

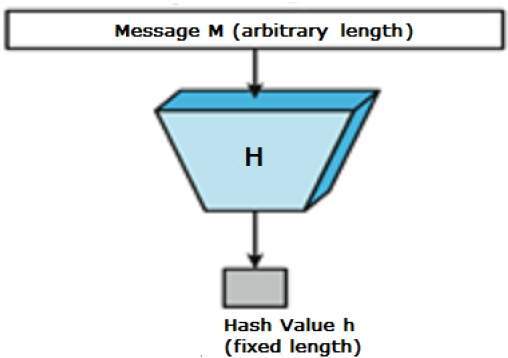
* **Keccak 256**
* **SHA 256**
* **Ripemd 160**
* **Ecrecover**

**Hash Functions**

Despite their simplicity, hash functions are incredibly valuable and may be found in nearly all information security applications.

A hash function is a mathematical function that turns a numerical input value into another numerical input value that has been compressed. When using the hash function, the input can be of any length, but the result is always of a set length.

The values provided by a hash function are referred to as message digests, or simply hash values, in some circles. The hash function is represented in the following diagram.

****

Characteristics of Hash Functions

The following are typical characteristics of hash functions:

Output with a Fixed Length (Hash Value)

* The hash function converts data of any length to data of a specific length. This method is referred to as hashing the data in some circles.
* As a rule, the hash function is significantly smaller than the input data, which is why hash functions are often referred to as compression functions in some circles.
* Because a hash is a condensed representation of a bigger piece of data, it is also referred to as a digest in some circles.
* An n-bit hash function is a hash function that has an output of n bits and is referred to as such. Popular hash functions give results ranging from 160 to 512 bits in length.

Increased Operational Efficiency

* In general, given any hash function h with input x, the computation of h(x) is a quick operation that takes little time.
* A symmetric encryption method is significantly slower than a computationally hash function.

**Characteristics of Hash Functions**

The following characteristics of a hash function are desired in order for it to be a successful cryptographic tool:

**Resistance to the Pre-Image**

This feature implies that reversing a hash function should be computationally difficult to accomplish.

To put it another way, if a hash function h produces a hash value z, then finding any input value x that hashes to z should be a tough operation.

This property protects against an attacker who only has a hash value and is attempting to decipher the input value from the hash value.

**Resistance to the Second Pre-Image**

This feature indicates that given an input and its hash, it should be difficult to identify a different input with the same hash in the same situation.

As a result of the fact that each given input value x produces the hash value h(x), it should be impossible to locate any other input value y for which the hash function (h) equals the value of the input value h(x) (x).

In this way, a malicious attacker who has access to an input value and its hash but wishes to replace a different value as legal value in lieu of the original input value is prevented from gaining access to the data.

**Resistance to Collisions**

This characteristic indicates that it should be difficult to identify two separate inputs of any length that result in the same hash value. This characteristic is regarded as a collision-free hash function in some circumstances.

To put it another way, for a hash function h, it is difficult to identify any two independent inputs x and y such that h(x) = h(y) (y).

Due to the fact that a hash function is a compression function with a defined hash length, it is impossible for a hash function to be completely free of collisions. This virtue of being collision free only serves to reinforce the notion that these collisions should be difficult to detect.

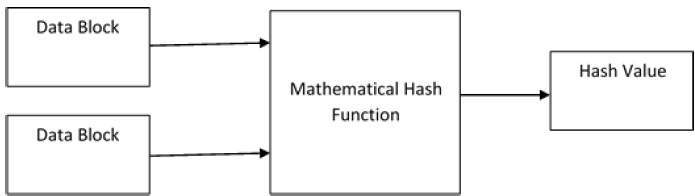
The fact that this feature exists makes it extremely difficult for an attacker to locate two input values that have the same hash value.

Furthermore, if a hash function is collision-resistant, it is also resistant to second pre-image collisions.

## Design of Hashing Algorithms

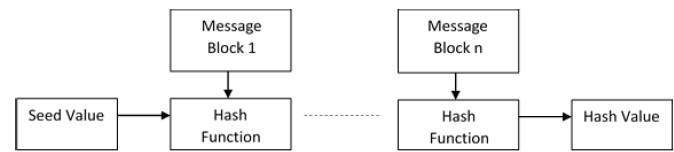
At the heart of a hashing is a mathematical function that operates on two fixed-size blocks of data to create a hash code. This hash function forms part of the hashing algorithm.

The size of each data block varies depending on the algorithm. Typically the block sizes are from 128 bits to 512 bits. The following illustration demonstrates hash function −



The hashing algorithm, which is similar to a block cypher, consists of rounds of the hash function described above. For each round, a fixed-size input is required, which is often a combination of the most recent message block and the output from the previous round.

This operation is repeated as many times as necessary to hash the entire message, up to the maximum number of rounds allowed. The following graphic depicts a schematic representation of the hashing algorithm:



As a result, the hash value of the first message block is used as an input for the second hash operation, the output of which influences the result of the third operation, and so forth. This phenomenon is referred to as the avalanche effect of hashing.

When two messages differ by even a single bit of data, the avalanche effect results in significantly different hash values for the two messages.

Understand the difference between a hash function and an algorithm in the proper manner. The hash function generates a hash code by performing operations on two blocks of binary data of defined length.

In computing, a hash algorithm is a procedure for employing the hash function that specifies how the message will be divided up and how the results from previous message blocks will be linked together.

Solidity includes many cryptographic functions that can be used to protect data.

The methods listed below are essential.

keccak256(bytes memory) returns (bytes32)- It computes the Keccak-256 hash of the given input.

ripemd160(bytes memory) returns (bytes20) -This function is used to compute the ripemd160 hash of the input compute using RIPEMD-160 algorithm.

sha256(bytes memory) returns (bytes32) - Computes the SHA-256 hash of the input using the sha256(bytes memory) returns (bytes32) function

ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address) - This function recovers the address associated with the public key from elliptic curve signature or returns zero if there is an error. The following parameters of the function correspond to the ECDSA values of the signature: r represents the first 32 bytes of the signature; s represents the second 32 bytes of the signature; and v represents the final 1 byte of the signature. This method returns an address in the form of a string.

The following function implements the keccak256,ripemd160 and sha256 algorithms:

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  contract hashFunc{  function haskWithKeccack(uint \_x,string memory name, address \_add) public returns(bytes32){  return keccak256(abi.encode(\_x,name,\_add));  }    function haskWithSHA256(uint \_x,string memory name, address \_add) public returns(bytes32){  return sha256(abi.encode(\_x,name,\_add));  }    function haskWithRipemd160(uint \_x,string memory name, address \_add) public returns(bytes32){  return ripemd160(abi.encode(\_x,name,\_add));  } } |
| --- |

The following code illustrates the use of ecrecover for verifying signed messages:

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  /\* Signature Verification  How to Sign and Verify # Signing 1. Create message to sign 2. Hash the message 3. Sign the hash (off chain, keep your private key secret)  # Verify 1. Recreate hash from the original message 2. Recover signer from signature and hash 3. Compare recovered signer to claimed signer \*/  contract VerifySignature {  /\* 1. Unlock MetaMask account  ethereum.enable()  \*/   /\* 2. Get message hash to sign  getMessageHash(  0x14723A09ACff6D2A60DcdF7aA4AFf308FDDC160C,  123,  "coffee and donuts",  1  )   hash = "0xcf36ac4f97dc10d91fc2cbb20d718e94a8cbfe0f82eaedc6a4aa38946fb797cd"  \*/  function getMessageHash(  address \_to,  uint \_amount,  string memory \_message,  uint \_nonce  ) public pure returns (bytes32) {  return keccak256(abi.encodePacked(\_to, \_amount, \_message, \_nonce));  }   /\* 3. Sign message hash  # using browser  account = "copy paste account of signer here"  ethereum.request({ method: "personal\_sign", params: [account, hash]}).then(console.log)   # using web3  web3.personal.sign(hash, web3.eth.defaultAccount, console.log)   Signature will be different for different accounts  0x993dab3dd91f5c6dc28e17439be475478f5635c92a56e17e82349d3fb2f166196f466c0b4e0c146f285204f0dcb13e5ae67bc33f4b888ec32dfe0a063e8f3f781b  \*/  function getEthSignedMessageHash(bytes32 \_messageHash)  public  pure  returns (bytes32)  {  /\*  Signature is produced by signing a keccak256 hash with the following format:  "\x19Ethereum Signed Message\n" + len(msg) + msg  \*/  return  keccak256(  abi.encodePacked("\x19Ethereum Signed Message:\n32", \_messageHash)  );  }   /\* 4. Verify signature  signer = 0xB273216C05A8c0D4F0a4Dd0d7Bae1D2EfFE636dd  to = 0x14723A09ACff6D2A60DcdF7aA4AFf308FDDC160C  amount = 123  message = "coffee and donuts"  nonce = 1  signature =  0x993dab3dd91f5c6dc28e17439be475478f5635c92a56e17e82349d3fb2f166196f466c0b4e0c146f285204f0dcb13e5ae67bc33f4b888ec32dfe0a063e8f3f781b  \*/  function verify(  address \_signer,  address \_to,  uint \_amount,  string memory \_message,  uint \_nonce,  bytes memory signature  ) public pure returns (bool) {  bytes32 messageHash = getMessageHash(\_to, \_amount, \_message, \_nonce);  bytes32 ethSignedMessageHash = getEthSignedMessageHash(messageHash);   return recoverSigner(ethSignedMessageHash, signature) == \_signer;  }   function recoverSigner(bytes32 \_ethSignedMessageHash, bytes memory \_signature)  public  pure  returns (address)  {  (bytes32 r, bytes32 s, uint8 v) = splitSignature(\_signature);   return ecrecover(\_ethSignedMessageHash, v, r, s);  }   function splitSignature(bytes memory sig)  public  pure  returns (  bytes32 r,  bytes32 s,  uint8 v  )  {  require(sig.length == 65, "invalid signature length");   assembly {  /\*  First 32 bytes stores the length of the signature   add(sig, 32) = pointer of sig + 32  effectively, skips first 32 bytes of signature   mload(p) loads next 32 bytes starting at the memory address p into memory  \*/   // first 32 bytes, after the length prefix  r := mload(add(sig, 32))  // second 32 bytes  s := mload(add(sig, 64))  // final byte (first byte of the next 32 bytes)  v := byte(0, mload(add(sig, 96)))  }   // implicitly return (r, s, v)  } } |
| --- |