# Intro to cryptographic hashing & modular arithmetic

by Richard K. Herz, www.ReactorLab.net, github.com/RichardHerz

---

## Cryptographic Hashing

Hashing consists of processing messages of arbitrary length to a unique hash value of fixed length such that any change in the message results in a large change in the hash value.

This result is valuable. You can verify that a message hasn't been tampered with by running the message through the hash algorithm and ensuring that the hash value hasn't changed from that supplied with the message. Even a tiny change in the message will result in a large and obvious change in the hash. In that case, you know that the message has been tampered with.

Hashing is used in blockchains in several ways: (1) create unique addresses for exchange of information (e.g., bitcoins), (2) create signatures of transactions for use in Merkle trees, which are incorporated in blocks in order to identify the transactions in a block of transactions, (3) to create a hash of an entire block for incorporation of that hash into the next block in the blockchain to prevent modification of previous blocks in the blockchain.

A block contains the hash of the previous block such that any change in any block changes the hashes of that and all following blocks.

Claude Shannon's information theory on cryptography states that the concepts of "confusion and diffusion" are key to hash functions.

"Confusion" is the result that the hash value has no apparent correspondence with the message. That is, by looking at the hash, you can't tell what was in the message.

"Diffusion" is that a change in any byte in the message will change most bytes in the hash at random locations. That is, a small change in the message produces large - and obvious - changes in the hash value.

Using modular arithmetic and "wrapping around" values at the end of the hash length produces a hash of constant length.

We wrote a program that executes the MD2 hash algorithm. This hash algorithm was published in 1989 by Ronald Rivest (the R in RSA encryption). The MD2 hash value length is 128 bits = 16 bytes. The most-used hash algorithm currently (2021) is the SHA-256 algorithm, which produces a 256-bit, 32-byte hash.

A brief explanation of modular arithmetic and a listing of a MD2 hash function are shown in two sections below.

The MD2 hash is no longer secure because of increased computer power but, since it is relatively simple, it is useful in learning about one method in which hashes provide “confusion” and “diffusion.”

A plain-text message is first converted to an integer by converting the message to a string of the ASCII or Unicode values of the characters in the message. The hash value, an integer, is usually displayed as a hexadecimal number.

In the MD2 hash, an array *S* of 256 bytes of random values is used.

“Confusion” is provided by the bitwise XOR (exclusive OR) of each message byte with a byte in the *S*-array of random numbers to generate a byte in the hash. The apparent randomness of the hash actually contains information from the message. An bitwise XOR operation at one bit location returns 1 if either, but not both, of the input bits is 1.

“Diffusion” is provided by using the XOR result at one byte location of the message to get the *S*-array value to use for the next message byte. This change propagates to the next byte, and so on. This results in a single change in the message affecting many bytes in the hash value. The hashing operations are repeated many times over the message such that a change even in the last character in the message will change the entire hash value.

Padding a message (adding extra bytes to it) so it's a multiple of the number of bytes in the hash allows both short and long messages to be hashed to a fixed length result by mod (hash length).

On the next page is an illustration of one operation in the MD2 hash, which is intended to show how “confusion” and “diffusion” is introduced into the hash.

## One step in one of the operations in the MD2 cryptographic hash (C array)

- "**Confusion**" by mixing message information with random numbers from the S-box
- "**Diffusion**" by value in current position in hash selecting S-box number for next message position such that confused information cascades and a change anywhere in message affects all hash locations through repeated passes through the message.

*The message*

| T | e | d | d | ··· | m | o | t | o |
|---|---|---|---|---|---|---|---|---|

text > UTF hexidecimal

| 54 | 65 | 64 | 64 | ··· | 6d | 6f | 74 | 6f |
|---|---|---|---|---|---|---|---|---|

info for step 4

binary values

| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|

XOR (bitwise exclusive OR)

| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

=

| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|

this message value is mixed with **random** value from S-box in hash from previous step, selected by operation with previous message value

info from step 3 for step 4

Substitution S-box (hexadecimal values)

| | | | |
|---|---|---|---|
| 29 | 2e | ··· | 13 |
| 62 | a7 | ··· | ca |
| : | : | ⋱ | : |
| 31 | 44 | ··· | 14 |

select value from element number hex c1 of S-box

value from element number hex c1 of S-box is hex ec

*The hash being computed*

| | | a5 | ec | ··· | | | | |
|---|---|---|---|---|---|---|---|---|

info from step 4 for step 5

S-box is array of 256 elements, each containing a **randomly** placed, non-repeating value in decimal range 0-255

MD2 hash length is 32 hex char = 16 each, 8-bit bytes = 128 bits, regardless of length of message, and maintained at this length by modular arithmetic

## Basic equations in modular arithmetic

All variables represent integers. The integers $a$ and $b$ are congruent modulo $n$.

$a \equiv b \pmod{n}$

where “mod” in parenthesis means that the modulus operation applies to all parts of the expression. This can also be expressed as

$a = b + kn$

That is, $a$ is divisible $k$ times by $n$, with remainder $b$.

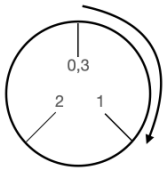
Those of us who use a 12 hour clock dial use modular arithmetic everyday. For example, the hour hand points to 6 on the dial at 6 am and 6 pm, where 6 pm is 18:00 in 24-hour time.

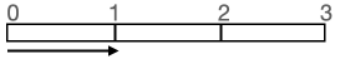
$18 \equiv 6 \pmod{12}$

$18 = 6 + 1(12)$

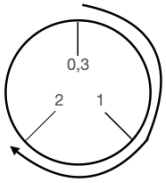
Cryptography and hashing use very large integers. For illustration here, we use small integers. For some operations below, we will use a dial or linear scale with three divisions to simplify things. The relationships shown work for numbers of all magnitudes.

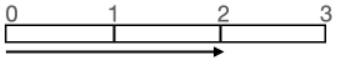
$1 \equiv 1 \pmod{3}$



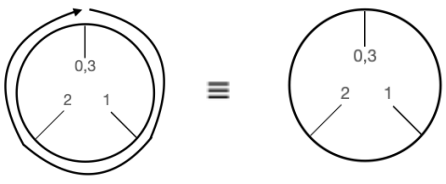



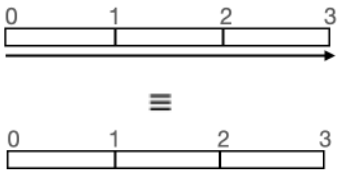

$2 \equiv 2 \pmod{3}$



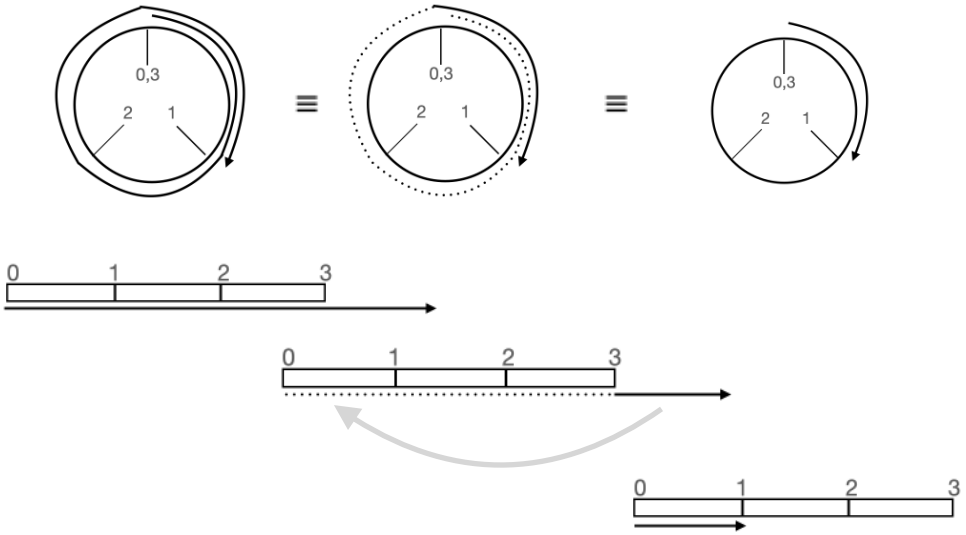




$3 \equiv 0 \pmod{3}$



$4 \equiv 1 \pmod 3$




One way to think of this is that numbers greater than some multiple $k$ of the modulus $n$ "wraps around" so that the maximum value in the system is $n$. In the examples above $k = 1$ but $k$ may have any integer value.

When we discuss hash algorithms, you will see that this "wrap around" feature is what keeps hash values a constant length, regardless of the length of the message being hashed.

One of the important relations in RSA cryptography is Fermat's Little Theorem.

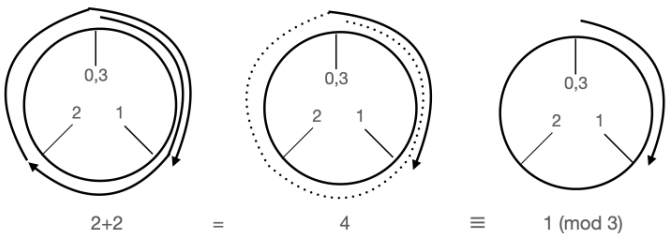
In the first relationship in the theorem, $m$ is an integer and $p$ is a prime number (prime integer)

$m^{(p-1)} \equiv 1 \pmod p$

As an example,

$2^{(3-1)} \equiv 1 \pmod 3$

$2^{(3-1)} = 2^2 = (2 + 2) \equiv 1 \pmod 3$

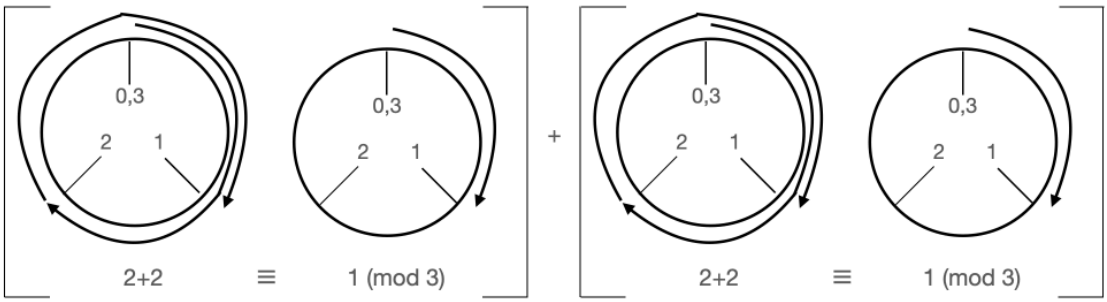


In the second relationship,

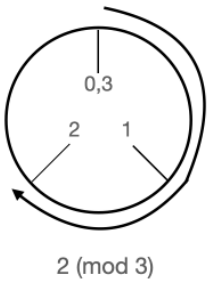
$m^p \equiv m \pmod{p}$

$2^3 \equiv 2 \pmod{3}$

$2^3 = 2(2^2) = (2 + 2) + (2 + 2) = 8 \equiv 2 \pmod{3}$




≡




Fermat's Little Theorem is a special case of the Euler-Fermat Theorem.

In that theorem, if the following apply,

$m^{(p-1)} \equiv 1 \pmod{p}$ and $m^{(q-1)} \equiv 1 \pmod{q}$

where $p$ and $q$ are prime numbers, then

$m^{(p-1)(q-1)} \equiv 1 \pmod{pq}$

This can be expressed as

$m^{\phi(n)} \equiv 1 \pmod{n}$

where $n = pq$

and $\phi$ is Euler's phi function

$\phi(n) = \phi(pq) = \phi(p)\phi(q) = (p - 1)(q - 1)$

## MD2 hash function in JavaScript

```
function fMD2(ms) {

  // returns MD2 hash of ms
  // by Richard K. Herz, github.com/RichardHerz, www.ReactorLab.net
  // implement MD2 hash in JavaScript
  // use algorithm from https://tools.ietf.org/html/rfc1319

  // 256-byte 'S table' from
  //     https://en.wikipedia.org/wiki/MD2_(hash_function)
  // S provides Shannon's "confusion" by mixing random info with the message
  // note: S[0] is returned as 41, which is the decimal equiv of hex 0x29
  // there are S[0] to S[255] = 256 values listed ranging in value from
  //     0x00 = 0 to 0xFF = 255 with no values repeated
  let S = [0x29, 0x2E, 0x43, 0xC9, 0xA2, 0xD8, 0x7C, 0x01, 0x3D, 0x36, 0x54, 0xA1, 0xEC, 0xF0, 0x06, 0x13,
           0x62, 0xA7, 0x05, 0xF3, 0xC0, 0xC7, 0x73, 0x8C, 0x98, 0x93, 0x2B, 0xD9, 0xBC, 0x4C, 0x82, 0xCA,
           0x1E, 0x9B, 0x57, 0x3C, 0xFD, 0xD4, 0xE0, 0x16, 0x67, 0x42, 0x6F, 0x18, 0x8A, 0x17, 0xE5, 0x12,
           0xBE, 0x4E, 0xC4, 0xD6, 0xDA, 0x9E, 0xDE, 0x49, 0xA0, 0xFB, 0xF5, 0x8E, 0xBB, 0x2F, 0xEE, 0x7A,
           0xA9, 0x68, 0x79, 0x91, 0x15, 0xB2, 0x07, 0x3F, 0x94, 0xC2, 0x10, 0x89, 0x0B, 0x22, 0x5F, 0x21,
           0x80, 0x7F, 0x5D, 0x9A, 0x5A, 0x90, 0x32, 0x27, 0x35, 0x3E, 0xCC, 0xE7, 0xBF, 0xF7, 0x97, 0x03,
           0xFF, 0x19, 0x30, 0xB3, 0x48, 0xA5, 0xB5, 0xD1, 0xD7, 0x5E, 0x92, 0x2A, 0xAC, 0x56, 0xAA, 0xC6,
           0x4F, 0xB8, 0x38, 0xD2, 0x96, 0xA4, 0x7D, 0xB6, 0x76, 0xFC, 0x6B, 0xE2, 0x9C, 0x74, 0x04, 0xF1,
           0x45, 0x9D, 0x70, 0x59, 0x64, 0x71, 0x87, 0x20, 0x86, 0x5B, 0xCF, 0x65, 0xE6, 0x2D, 0xA8, 0x02,
           0x1B, 0x60, 0x25, 0xAD, 0xAE, 0xB0, 0xB9, 0xF6, 0x1C, 0x46, 0x61, 0x69, 0x34, 0x40, 0x7E, 0x0F,
           0x55, 0x47, 0xA3, 0x23, 0xDD, 0x51, 0xAF, 0x3A, 0xC3, 0x5C, 0xF9, 0xCE, 0xBA, 0xC5, 0xEA, 0x26,
           0x2C, 0x53, 0x0D, 0x6E, 0x85, 0x28, 0x84, 0x09, 0xD3, 0xDF, 0xCD, 0xF4, 0x41, 0x81, 0x4D, 0x52,
           0x6A, 0xDC, 0x37, 0xC8, 0x6C, 0xC1, 0xAB, 0xFA, 0x24, 0xE1, 0x7B, 0x08, 0x0C, 0xBD, 0xB1, 0x4A,
           0x78, 0x88, 0x95, 0x8B, 0xE3, 0x63, 0xE8, 0x6D, 0xE9, 0xCB, 0xD5, 0xFE, 0x3B, 0x00, 0x1D, 0x39,
           0xF2, 0xEF, 0xB7, 0x0E, 0x66, 0x58, 0xD0, 0xE4, 0xA6, 0x77, 0x72, 0xF8, 0xEB, 0x75, 0x4B, 0x0A,
           0x31, 0x44, 0x50, 0xB4, 0x8F, 0xED, 0x1F, 0x1A, 0xDB, 0x99, 0x8D, 0x33, 0x9F, 0x11, 0x83, 0x14];

  let hb = 16; // hexadecimal radix (base)
  let md = []; // initialize decimal representation of message
  let cd;

  // convert message string from ASCII char to ASCII decimal array
  // do all processing in decimal, then, at end, convert hash to hex string
  for (let i in ms) {
    cd = ms.charCodeAt(i);
    md.push(cd);
  }

  // "Step 1. Append Padding Bytes" ----------------------

  // append "i" bytes of value "i" to get mod 0
  // but add at least one byte
  let mdl = md.length;
  let mdlmod = mdl % hb; // % is the modulo operator
  let p = hb - mdlmod;
  for (let i = 0; i < p; i++) {
    md.push(p);
  }
```

```
// "Step 2. Append Checksum" ----------------------------

// clear checksum C
let C = [];
for (let i = 0; i < hb; i++) {
  C[i] = 0;
}
let L = 0;
// get new length, changed after padding
mdl = md.length;
let N = mdl / hb;

for (let i = 0; i < N; i++) {
  for (let j=0; j < hb; j++) {
    // see  https://decimaltobinary.pro/
    // ^ is bitwise XOR in JavaScript
    C[j] = S[md[i*hb+j] ^ L];          <<< These two lines are illustrated
    L = C[j];                                 by the graphic on page 3
  }
}


// append C to message
md = md.concat(C);


// "Step 3. Initialize MD Buffer" -----------------------

// "A 48-byte buffer X is used to compute the message digest. The buffer
// is initialized to zero."
// note 48 = 3*16, so X is three 16-byte blocks
// the first 16-byte block ends up being the final MD2 hash

let X = [];
for (let i = 0; i < 48; i++) {
  X[i] = 0;
}

// "Step 4. Process Message in 16-Byte Blocks" ----------

// get new length, changed after appending checksum C
mdl = md.length;
N = mdl / hb;

// "Process each 16-byte block"
for (let i = 0; i < N; i++) {

  // "Copy block i into X"
  //
  // "For j = 0 to 15 do
  //    Set X[16+j] to M[i*16+j].
  //    Set X[32+j] to (X[16+j] xor X[j]).
  //  end"
```

```
    for (let j = 0; j < hb; j++) {
      X[hb+j] = md[i*hb+j];
      X[2*hb+j] = X[hb+j] ^ X[j];
    }

    // "Set t to 0"
    let t = 0;

    // "Do 18 rounds."
    for (let j = 0; j < 18; j++) {

      // "For k = 0 to 47 do
      //    Set t and X[k] to (X[k] xor S[t]).
      // end /* of loop on k */"

      for (let k = 0; k < 48; k++) {
        X[k] = X[k] ^ S[t];
        t = X[k];
      } // end of loop on k

      // "Set t to (t+j) modulo 256"

      t = (t+j) % 256; // % is the modulo operator

    } // end of loop on j
  } // end of loop on i

  // "Step 5. Output" ------------------------------------

  // "The message digest produced as output is X[0 ... 15]. That is, we
  //  begin with X[0], and end with X[15]"

  let MD2d = X.slice(0,hb);
  let MD2 = '';
  let ch;
  // convert decimal MD2d array to hex string MD2
  for (let i = 0; i < hb; i++) {
    ch = MD2d[i].toString(hb);
    if (MD2d[i] < hb ) {
      ch = '0' + ch;
    }
    MD2 += ch;
  }

  return MD2;
} // END of fMD2
```