Cryptographical Algorithms

Basic concepts

Cryptography (the term) comes from the Greek words (κρυπτός) which means “hidden” and γράφειν which means “writing”. So, it means cipher.

The topic itself has become independent over the years, even though it was always part of the computer science / information technology discipline and its major task was to study and decipher secret codes.

With the widespread nature of electronic communication three major basic requirements appeared in the method and use. By them we can successfully build an online communication channel:

The requirements of a secure channel:

Secrecy: Apart from the parties which are communicating with each other nobody else should see the messages.

Cryptographic algorithm. Our main goal in this task is to provide a method which is uncrackable, easy to use and calculate at the same time.

The limits of this method must be demonstrated/proved mathematically.

Credibility: The communicating parties can ascertain the identity of each other (even though they have never met in real life)

Tools: Authentication services (authorities) It is a natural demand in the case of online communication that our identity must be protected by an algorithm even though we must use such remote servers that cannot be traced down or approached.

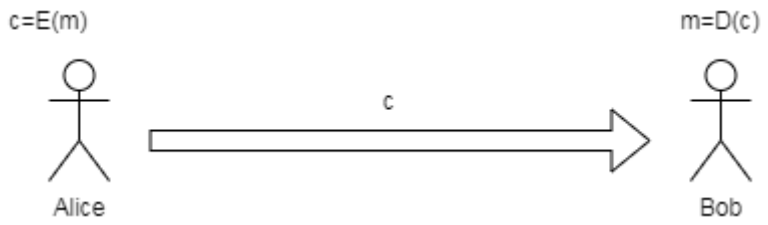
There is no 100% security, but even in this case the boundaries must be known.

Integrity: any kind of undetected alteration of our data must be blocked.

Digital signature. Nowadays it is common that we must sign online contracts where the signatories to the treaty cannot be present physically. Even the signatures happen in different times.

The use of digital signatures is now a mandatory area in Hungary too, for example in the case of court administration.

The following figure describes the basic model of communication, and its notations.



Alice wants to send a message denoted as m to Bob. The encoded message is c.

Function E() is called as encrypt/encoding function

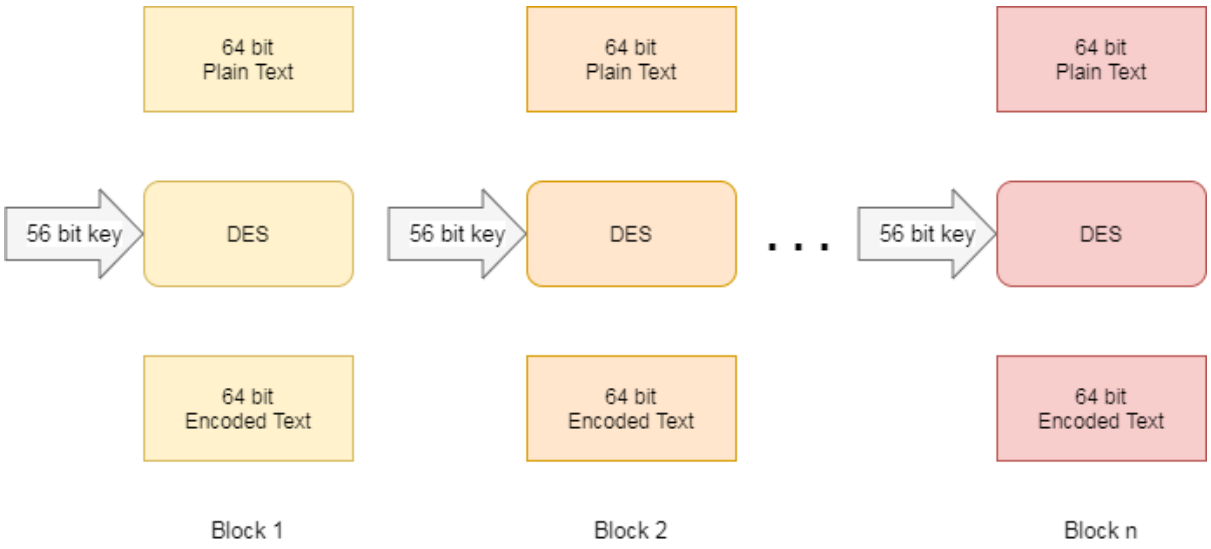
Function D() is called decrypt/decoding function.

DES (Data Encryption Standard)

DES is a widespread encryption algorithm. It was developed by IBM in early 1970's and then registered by National Bureau of Standards [[1]]

DES is a symmetric block cypher; the algorithm takes a fixed length 64 bit long plain text. The output of the algorithm is of the same size, DES uses a key which has a different bitsize: 56 bits.

The key however appears to be 64 bit long, but the algorithm ignores every 8th bit.



For the shake of simplicity, the 64 bit long block or key will be represented as 16 hexadecimal numbers.

Steps

1. A 64 bit long chunk of the plain text message is given to the initial permutation function for execution

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

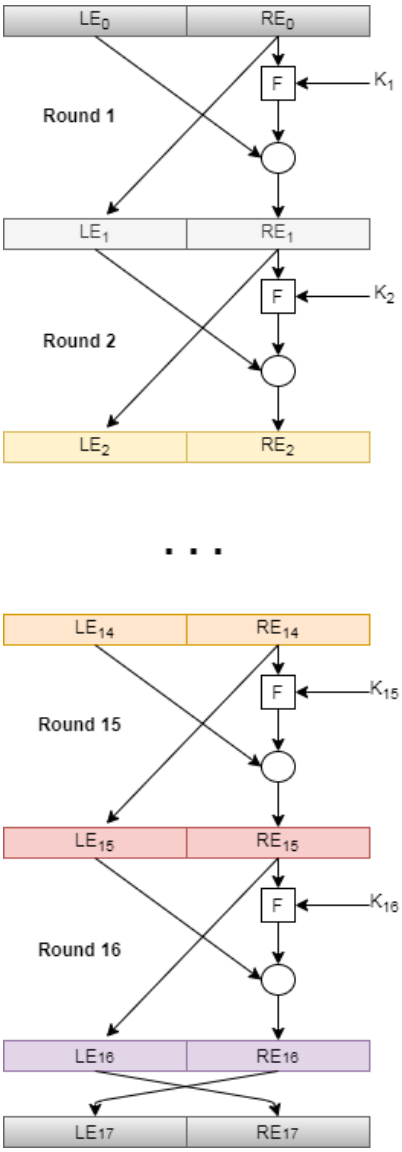
   The 58th bit of the plain text becomes the 1st bit, 50th bit will be the 2nd, … and the 7th bit will be the 64th bit

2. The permuted part splits the data into two 32 bit parts: left plain text (LFT) and right plain text part (RFT)
3. Both undergo a 16 round of encryption process using the 56 bit key

4. The two parts then rejoined, and a final permutation is performed on the combined block, so the 64 bit cypher block is generated. The final permutation is the inverse if the initial permutation:

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

5. The 16 round of encryption looks as follows

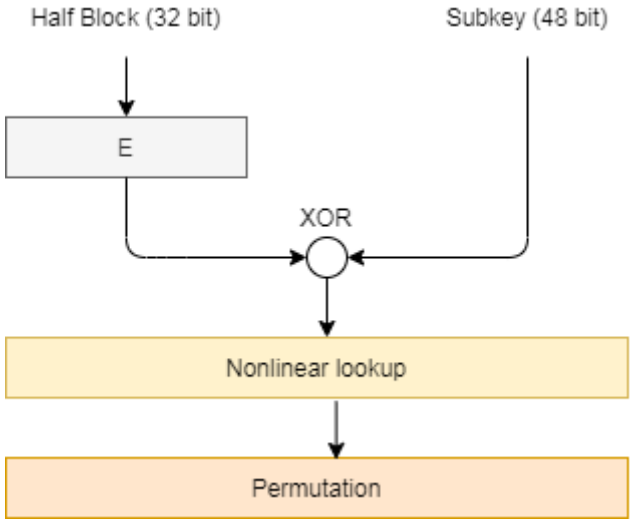


Each round generates the key K_i out of the 56-bit initial key. The key is divided into 28 bit long halves. In each round both halves are shifted left. In round 1, 2, 9 and 16 they are shifted by 1 bit, in other rounds two bits. Then a 48 bit subkey (K_i)is selected by a compression and permutation operation.

$LFT_n = RFT_{n-1}$

$RFT_n = LFT_{n-1}$ XOR $f(R_{n1}, K_n)$

The calculation function f is described as follows: In each round the 32 bit block is expanded to 48 bit using the expansion permutation, which duplicates half of the bits. Then the result is combined with the subkey by an XOR operation. The 48 bit long result then substitutes some bits according to a nonlinear lookup table. Finally, the output is rearranged by a fixed permutation.



Examples available at https://sandilands.info/crypto/DataEncryptionStandard.html#x16-840008.5
DES in OPEN SSL

AES (Advanced Encryption Standard)

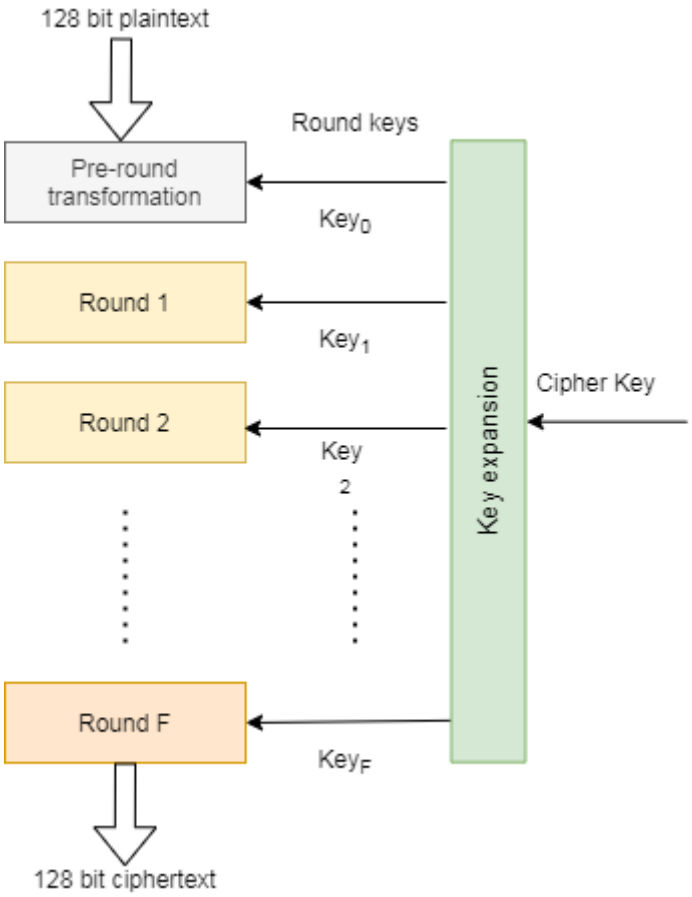
The Advanced Encryption Standard (AES) encryption specification was introduced by [[9]]. This algorithm as faster and more reliable than DES so became a US standard in 2001. It operates on 128 bit data, the key can be 128, 182 or 256 bit long. it is a symmetric key cipher.

AES is based on a substitution-permutation network. Some of its linked operations replaces inputs by specific outputs, some of them permutates the data.

The number of encryption rounds is based on the key size:

- 128 bits – 10 rounds
- 192 bits – 12 rounds
- 256 bit keys – 14 rounds

Each round uses its own key K_i calculated from the original key, using the AES key schedule. [[9]]



The relation between the number of rounds (N) and cipher key size is defined by the following table:

N	**Key size**
10	128
12	192
13	256

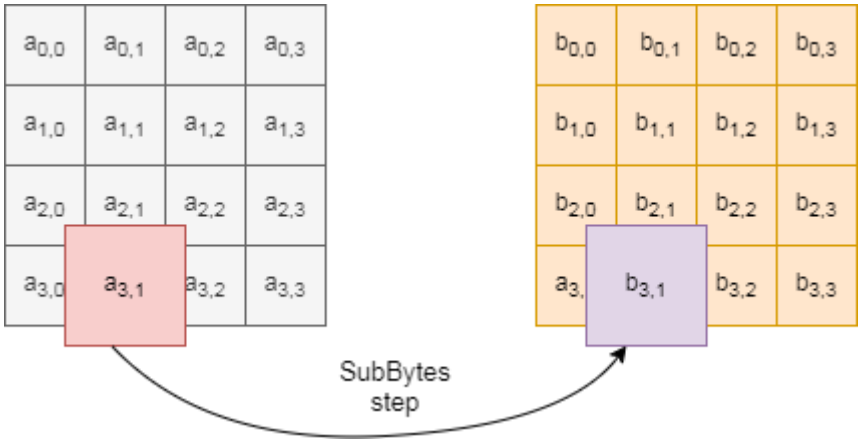
Description of the algorithm

KeyExpansion – round key K_i is derived from the cipher key using the AES key schedule Initial round key addition:

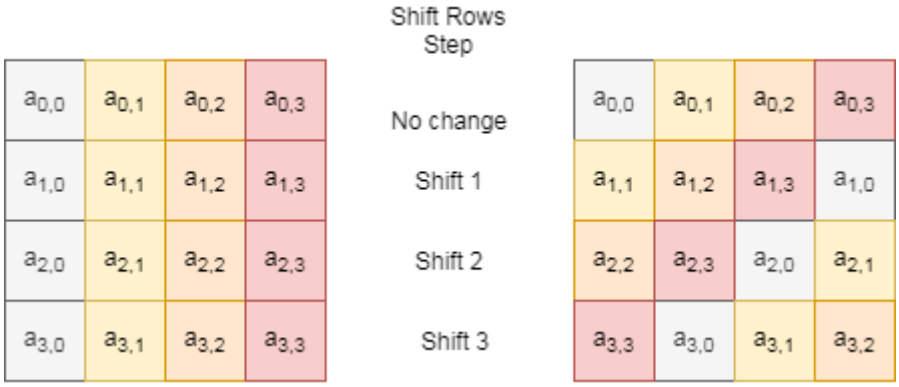
AddRoundKey – each byte of the state is combined with a byte of the round key using bitwise XOR.

9, 11 or 13 rounds:

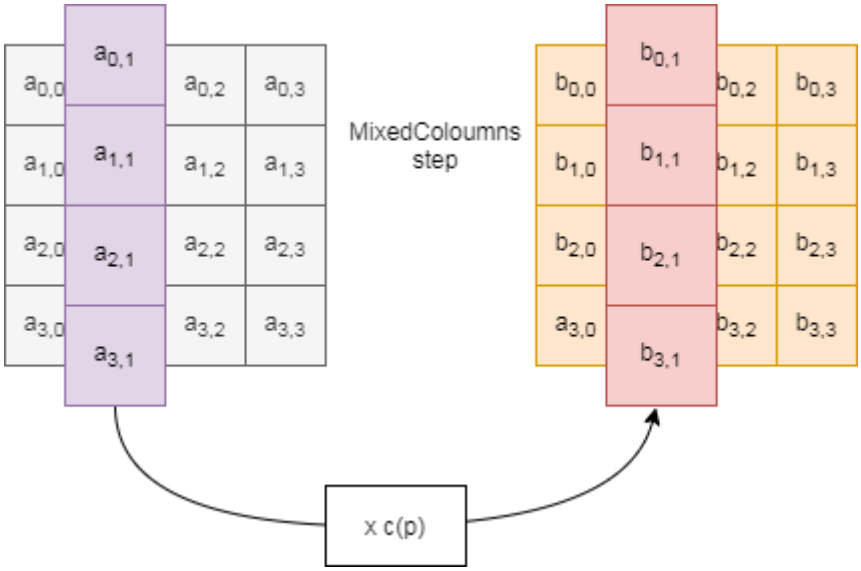
SubBytes – a non-linear substitution step where each byte is replaced with another according to a lookup table.



ShiftRows – a transposition step. The data is arranged to a 4 x 4 matrix, consisting of row r_0, r1, r_2, r_3. Each row r_i shifts I position left.



MixColumns – a linear transformation operation which operates on the columns of the matrix.



Attacks

https://en.wikipedia.org/wiki/Advanced_Encryption_Standard#cite_note-fips-197-6

RSA algorithm

Rivest, Shanir and Adleman introduced their asymmetric cryptographic algorithm in 1977 [[7]]. The algorithm is based on exponentiation and modulo. Let us assume that if the following equation is true:

$T^{ed} \bmod N = T$

in some special cases the equation can be decomposed into two pairs, one will encode and the second decode, as follows:

$T^{e} \bmod N = C$

$C^{d} \bmod N = T$

Unfortunately, this equation will not work with arbitrary triplets: (e,d,N)

Theorem

Let's originate from this old Greek equation:

$T^{N-1} \bmod N = 1$

where N > T and N is a prime number

Primes are integer numbers whose do not have integer divisors, like 11, 13, 19, ...

Let f(N) be a function called *totient* that indicates the number of positive integers up to N. Two integers are coprime if they share no common positive factors (divisors) except 1.

For example f(9) = 6 as 9 has 6 relative primes {1, 2, 4, 5, 7, 8}.

If N is a prime number, then calculation is straightforward. There is no integer number up to N that shares a common divisor, thus

f(N)+ 1 = N => N-1 = f(N)

If the exponent K is multiplied by a constant, then modulo will not be changed, so the following equation is still true

$T^{K\,f(N)} \bmod N = 1$

This step ensures that there are theoretically infinity number of keys as K is an arbitrary number. Let's multiply each side by T:

$T^{K\,f(N)+1} \bmod N = T$

Let’s select e and d keys so that

K f(N)+1 = e d

RSA key generation

Key generation can be carried out as follows. Let us have two random integer prime numbers: X and Y. Their product is N.

N = X Y

We know the how many relative primes X and Y have

f(X) = X-1

f(Y) = Y-1

f(N) can be calculated as

f(N) = (X-1) (Y-1)

Decompose K f(N)+1 into a product of two integer numbers

K f(N)+1 = e d

In practice using the following equation can be used for the decomposition. Select *e* so that

gcd(e, f(N)) = 1

where gcd stands for the greatest common divisor, and choose a d so that

1 < d < f(N)

and

e d mod f(N) = 1

public key (e,N), private key (d,N)

A working example

Let's choose two prime numbers: 67 and 11

N = 67 11 = 737

f(n) = 66 10 = 660

Choose e so that gcd(e, 660) = 1, let e be 7. So, the public key will be:

(N, e) = (737, 7)

Calculate d:

Find a K value so that K f(N)+1 = e d is true, in other words K f(N)+1 mod e =0

K*660 + 1 mod 7 = 0

K = 3 works, so

d = (3 * 660 +1) / 7 = 283

So private key is

(N, d) = (737. 283)

Encode the number 28

28^7 mod 737 = 316

Decode the number 316

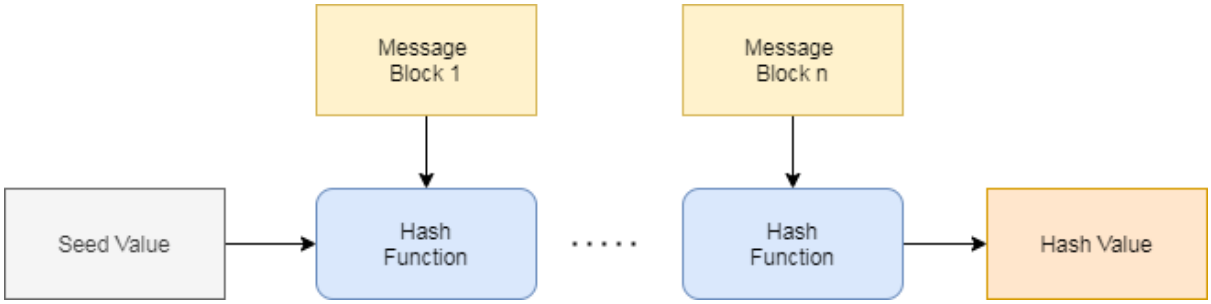
$316^{283} \bmod 737 = 28$

Hash functions

Hash function is a mathematical function which take inputs of variable lengths and returns a fixed length output.

The preferred hash function is

- easy to calculate for any data
- difficult to compute the plain text of a given hash
- different plain text of any length must not have the same hash
- dispersion property is preferred: small change in the plain text should result big changes in the hash



Usually plain text processed in data blocks. The output of one block will be the input of the following hash calculation step and so on. This called as an avalanche effect of hashing

Message-digest (MD5)

This message-digest algorithm is a widely used hash function producing a 128-bit hash value.
The plain text message is split into chunks of 512-bit blocks.
The plain text message is padded so that its length to be divisible by 512.

RFC1321 specifies padding instruction as follows:

3.1 Step 1. Append Padding Bits

The message is "padded" (extended) so that its length (in bits) is congruent to 448, modulo 512. That is, the message is extended so that it is just 64 bits shy of being a multiple of 512 bits long. Padding is always performed, even if the length of the message is already congruent to 448, modulo 512.

Padding is performed as follows: a single "1" bit is appended to the message, and then "0" bits are appended so that the length in bits of the padded message becomes congruent to 448, modulo 512. In all, at least one bit and at most 512 bits are appended.

3.2 Step 2. Append Length

A 64-bit representation of b (the length of the message before the padding bits were added) is appended to the result of the previous step. In the unlikely event that b is greater than 2^{64}, then only the low-order 64 bits of b are used. (These bits are appended as two 32-bit words and appended low-order word first in accordance with the previous conventions.)

At this point the resulting message (after padding with bits and with b) has a length that is an exact multiple of 512 bits. Equivalently, this message has a length that is an exact multiple of 16 (32-bit) words. Let M[0 ... N-1] denote the words of the resulting message, where N is a multiple of 16.

The main MD5 algorithm operates on 128-bit chunks which are first divided into four 32 bit words (A, B, C, D)

The processing of a message block consists of four rounds; each round is composed of 16 similar operations based on a non-linear function, modular addition, and left rotation.

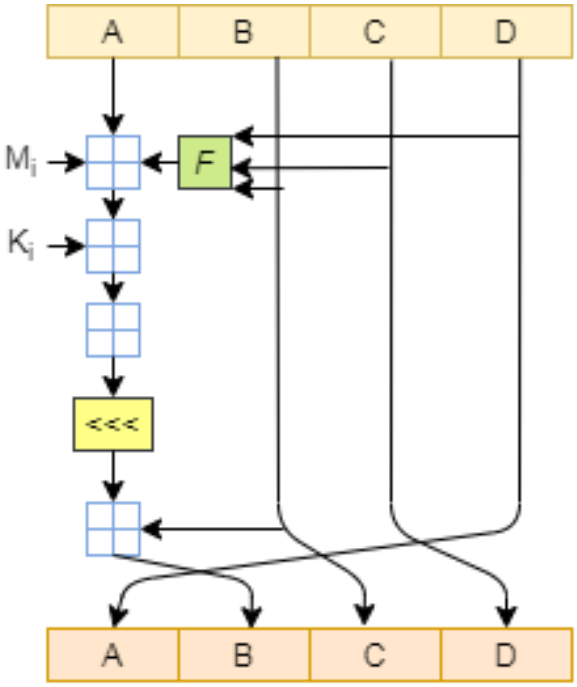
There are four possible functions; a different one is used in each round:

F(B, C, D) = (B and C) or (not B and D)

G(B, C, D) = (B and C) or (C and not D)

H(B, C, D) = B xor C xor D

I(B, C, D) = C xor (B or not D)




```
"""
The implementation of the MD5 algorithm is based on the original RFC at
https://www.ietf.org/rfc/rfc1321.txt and contains optimizations from
https://en.wikipedia.org/wiki/MD5.
"""

import struct
from enum import Enum
from math import (
    floor,
    sin,
)

from bitarray import bitarray
```

```
class MD5Buffer(Enum):
    A = 0x67452301
    B = 0xEFCDAB89
    C = 0x98BADCFE
    D = 0x10325476


class MD5(object):
    _string = None
    _buffers = {
        MD5Buffer.A: None,
        MD5Buffer.B: None,
        MD5Buffer.C: None,
        MD5Buffer.D: None,
    }

    @classmethod
    def hash(cls, string):
        cls._string = string

        preprocessed_bit_array = cls._step_2(cls._step_1())
        cls._step_3()
        cls._step_4(preprocessed_bit_array)
        return cls._step_5()

    @classmethod
    def _step_1(cls):
        # Convert the string to a bit array.
        bit_array = bitarray(endian="big")
        bit_array.frombytes(cls._string.encode("utf-8"))

        # Pad the string with a 1 bit and as many 0 bits required such that
        # the length of the bit array becomes congruent to 448 modulo 512.
        # Note that padding is always performed, even if the string's bit
        # length is already conguent to 448 modulo 512, which leads to a
        # new 512-bit message block.
        bit_array.append(1)
        while bit_array.length() % 512 != 448:
            bit_array.append(0)

        # For the remainder of the MD5 algorithm, all values are in
        # little endian, so transform the bit array to little endian.
        return bitarray(bit_array, endian="little")

    @classmethod
    def _step_2(cls, step_1_result):
        # Extend the result from step 1 with a 64-bit little endian
        # representation of the original message length (modulo 2^64).
        length = (len(cls._string) * 8) % pow(2, 64)
        length_bit_array = bitarray(endian="little")
        length_bit_array.frombytes(struct.pack("<Q", length))

        result = step_1_result.copy()
        result.extend(length_bit_array)
        return result
```

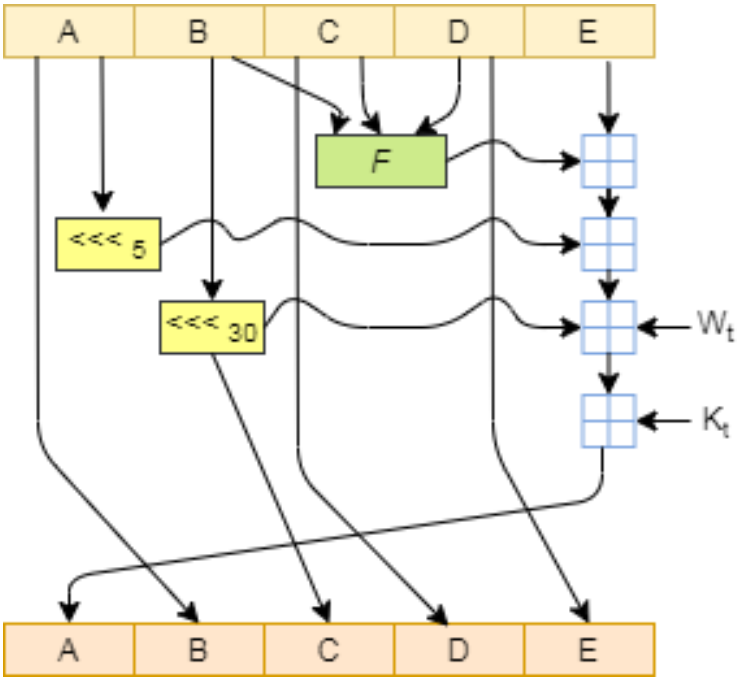
```
@classmethod
def _step_3(cls):
    # Initialize the buffers to their default values.
    for buffer_type in cls._buffers.keys():
        cls._buffers[buffer_type] = buffer_type.value

@classmethod
def _step_4(cls, step_2_result):
    # Define the four auxiliary functions that produce one 32-bit word.
    F = lambda x, y, z: (x & y) | (~x & z)
    G = lambda x, y, z: (x & z) | (y & ~z)
    H = lambda x, y, z: x ^ y ^ z
    I = lambda x, y, z: y ^ (x | ~z)

    # Define the left rotation function, which rotates `x` left `n` bits.
    rotate_left = lambda x, n: (x << n) | (x >> (32 - n))

    # Define a function for modular addition.
    modular_add = lambda a, b: (a + b) % pow(2, 32)

    # Compute the T table from the sine function. Note that the
    # RFC starts at index 1, but we start at index 0.
    T = [floor(pow(2, 32) * abs(sin(i + 1))) for i in range(64)]

    # The total number of 32-bit words to process, N, is always a
    # multiple of 16.
    N = len(step_2_result) // 32

    # Process chunks of 512 bits.
    for chunk_index in range(N // 16):
        # Break the chunk into 16 words of 32 bits in list X.
        start = chunk_index * 512
        X = [step_2_result[start + (x * 32) : start + (x * 32) + 32] for x in range(16)]

        # Convert the `bitarray` objects to integers.
        X = [int.from_bytes(word.tobytes(), byteorder="little") for word in X]

        # Make shorthands for the buffers A, B, C and D.
        A = cls._buffers[MD5Buffer.A]
        B = cls._buffers[MD5Buffer.B]
        C = cls._buffers[MD5Buffer.C]
        D = cls._buffers[MD5Buffer.D]

        # Execute the four rounds with 16 operations each.
        for i in range(4 * 16):
            if 0 <= i <= 15:
                k = i
                s = [7, 12, 17, 22]
                temp = F(B, C, D)
            elif 16 <= i <= 31:
                k = ((5 * i) + 1) % 16
                s = [5, 9, 14, 20]
                temp = G(B, C, D)
            elif 32 <= i <= 47:
                k = ((3 * i) + 5) % 16
                s = [4, 11, 16, 23]
```

```
                temp = H(B, C, D)
            elif 48 <= i <= 63:
                k = (7 * i) % 16
                s = [6, 10, 15, 21]
                temp = I(B, C, D)

            # The MD5 algorithm uses modular addition. Note that we need a
            # temporary variable here. If we would put the result in `A`, then
            # the expression `A = D` below would overwrite it. We also cannot
            # move `A = D` lower because the original `D` would already have
            # been overwritten by the `D = C` expression.
            temp = modular_add(temp, X[k])
            temp = modular_add(temp, T[i])
            temp = modular_add(temp, A)
            temp = rotate_left(temp, s[i % 4])
            temp = modular_add(temp, B)

            # Swap the registers for the next operation.
            A = D
            D = C
            C = B
            B = temp

        # Update the buffers with the results from this chunk.
        cls._buffers[MD5Buffer.A] = modular_add(cls._buffers[MD5Buffer.A], A)
        cls._buffers[MD5Buffer.B] = modular_add(cls._buffers[MD5Buffer.B], B)
        cls._buffers[MD5Buffer.C] = modular_add(cls._buffers[MD5Buffer.C], C)
        cls._buffers[MD5Buffer.D] = modular_add(cls._buffers[MD5Buffer.D], D)

    @classmethod
    def _step_5(cls):
        # Convert the buffers to little-endian.
        A = struct.unpack("<I", struct.pack(">I", cls._buffers[MD5Buffer.A]))[0]
        B = struct.unpack("<I", struct.pack(">I", cls._buffers[MD5Buffer.B]))[0]
        C = struct.unpack("<I", struct.pack(">I", cls._buffers[MD5Buffer.C]))[0]
        D = struct.unpack("<I", struct.pack(">I", cls._buffers[MD5Buffer.D]))[0]

        # Output the buffers in lower-case hexadecimal format.
        return f"{format(A, '08x')}{format(B, '08x')}{format(C, '08x')}{format(D, '08x')}"
```

SHA

Secure Hash Algorithm (SHA-1) is a cryptographic hash function. It produces a 160 bit hash of the data. Generally, this message digest is represented as a hexadecimal number.

In the beginning, we have 160 bits input, we break it down into 5 parts which we name to be A, B, C, D and E.



An iteration of SHA algorithm consist of the following:

- A, B, C, D, and E are 32-bit words
- F is a nonlienar function that varies
- $<<<_x$ denotes a left circular shift by x places, x varies
- W_t is the expanded message word of eound t
- K_t is a round constant of t
- ⊞ denotes addition modulo 2^{32}

For every 20 rounds, F_i and K_i are constant they have a set of predefined values and function description which remains common as follows:

Rounds 1-20

F_i = (B and C) or ((not B) and D)

K_i = 0x5A827999

Rounds 21-40

F_i = B xor C xor D

K_i = 0x6ED9EBA1

Rounds 41-60

F_i = (B and C) or (B and D) or (C and D)

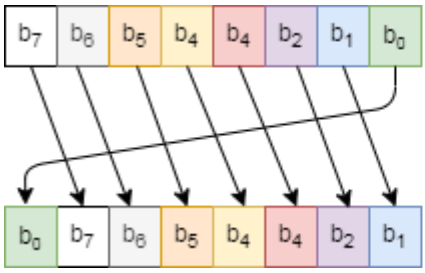
K_i = 0x8F1BBCDC

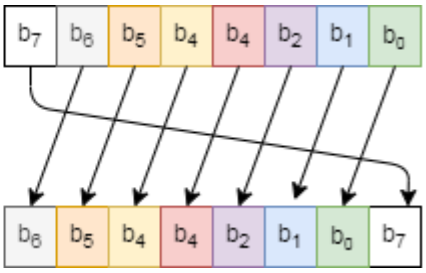
Rounds 61-80

F_i = B xor C xor D

K_i = 0xCA62C1D6

Circular shift operation operates on X bit long words. The following figure depicts left-shift and right-shift operation by 1 bit.





Creating expanded message

1. The message is then padded by appending a 1, followed by enough 0s until the message is 448 bits. The length of the message represented by 64 bits is then added to the end, producing a message that is 512 bits long
2. The padded input obtained above, MM, is then divided into 512-bit chunks, and each chunk is further divided into sixteen 32-bit words, $W_0 \ldots W_{15}$
3. For each chunk, begin the 80 iterations, i_i, necessary for hashing (80 is the determined number for SHA-1), and execute the following steps on each chunk
4. the following operation is performed:
   $W_i = S^1 (W_{i-3}$ XOR W_{i-8} XOR W_{i-14} XOR $W_{i-16})$
   S is the shift operator
5. Store the hash values defined in step 1 in the following variables:
   $A = H_0$
   $B = H_1$
   $C = H_2$
   $D = H_3$
   $E = H_4$
6. For iteration 80 compute
   $TEMP = S^5 (A) + F(B;C;D)+E+W_i+K_i$
   E=D
   D=C
   $C=S^{30}(B)$
   B=A
   A=TEMP
7. Store the result of the chunk's hash as follows
   $H_0 = H_0+A$
   $H_1 = H_1+B$
   $H_2 = H_2+C$
   $H_3 = H_3+D$
   $H_4 = H_4+E$
8. The final step is to compute the 160-bit message digest
   $H = S^{128}(H_0)$ OR $S^{96}(H_1)$ OR $S^{64}(H_2)$ OR $S^{32}(H_3)$ OR H_4
9. Message Authentication Code

Message Authentication Code

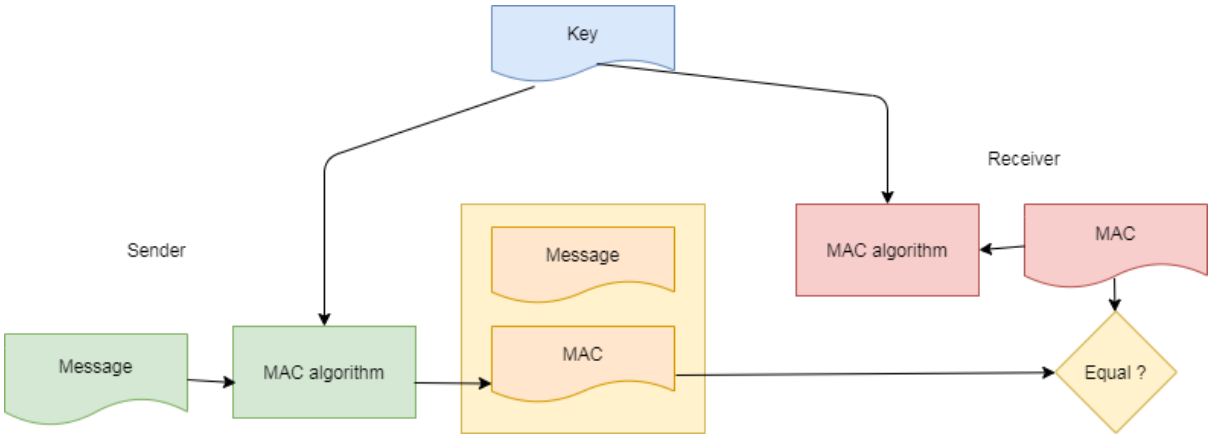
Let's overview the terminology. Message Authentication Code (MAC) are also called cryptographic checksums. MAC algorithm is a symmetric key algorithm to provide message authentication. Both the sender and receiver share the same key.

Basically, a MAC is an encrypted checksum of the plain text message. It is attached to the message to ensure its authentication. The receiver regenerates the MAC of the message received. If the computed MAC matches, then the message can be accepted. If the computed MAC does not match, then it is not possible to determine whether the message or the origin is genuine.

MAC like the hash functions generate a fixed length output of the input field. The difference is that MAC uses a secret key for the output generation. Main characteristic of the process is that MAC algorithm is public, the MAC value is produced by the secret key.

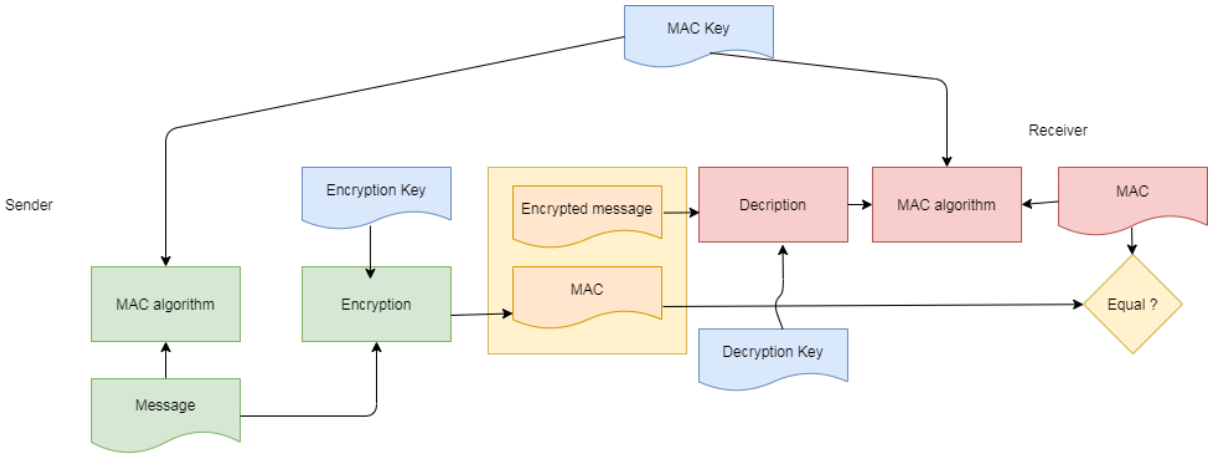
The main limitations are as follows:

- participant must be known in advance
- secret key must be established and shared in advance
- MAC can not proof that that a message was sent by the Sender
- it is not possible to determine which party generated the MAC



MAC with internal error code

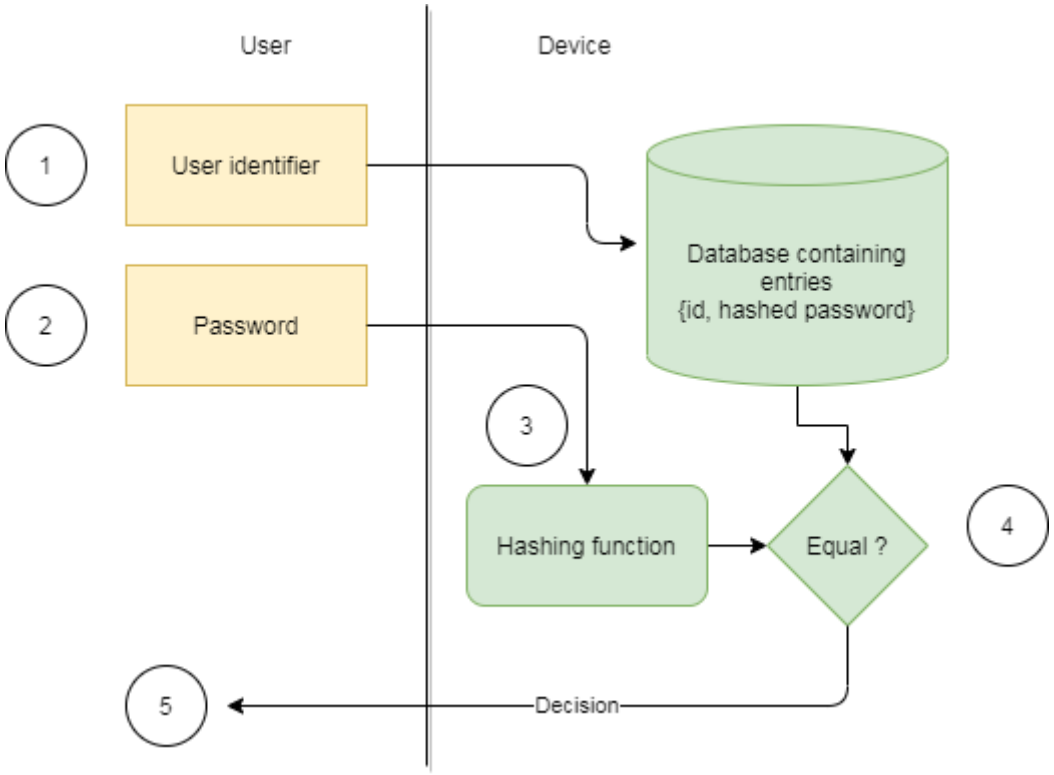
Sender can encrypt the content before sending it through the network. In this way the message will gain confidentiality.



Password storage

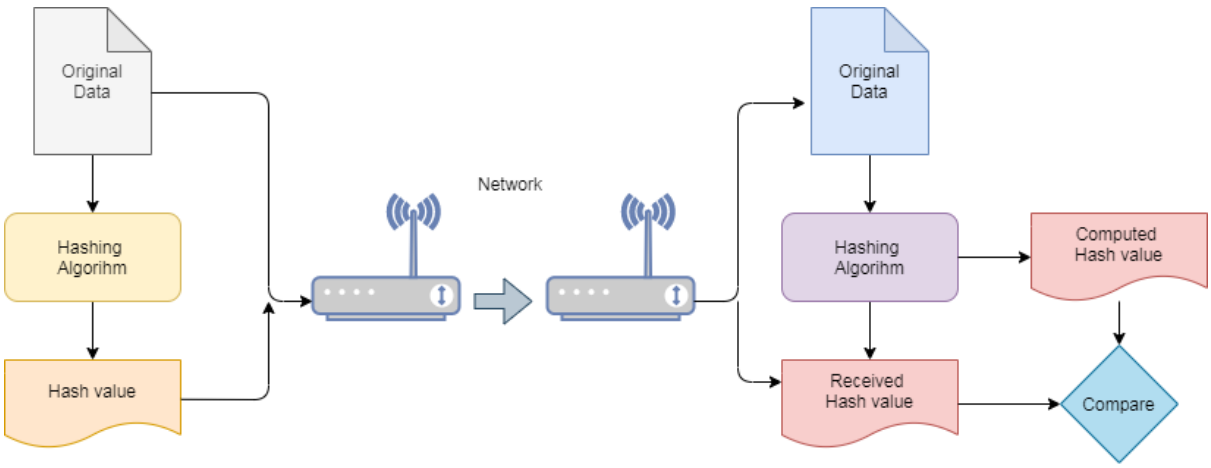
Hash functions provide protection to password storage. Rather than storing password as plain text logon process stores the hash values of passwords in the database.

The Password file consists of a table of pairs which are in the form (user_identifier, hash(PASSWORD)).



Data Integrity Check

H ash functions are widely used to check data integrity. The hash function is used to generate the checksums of the data



If the original data is modified, then checksum will not match. However, if an attacker can modify both the data and its checksum then the originality can not be checked.

Exercise

Password File Location and Content in Kali linux

Kali stores password data in file /etc/shadow. Only root user can write the file. The file stores username, hashed password, password change date, expiry date etc. in colon (:) separated format.

To query the file enter

sudo cat /etc/shadow

kali:6GHNiMeXhVU70giNl$vpm87wq/tB5X5rA8MYnsw8ssB7iyW.9gh5m/drfTmMJdvRtArB/3Xtyan1/DmOeBdpxs9cfKaDt0n15nqupvn/:18583:0:99999:7:::

Value	Meaning
6	Value between starting two $ sign represents algorithm used for hashing. Here number 6 suggests sha-512 been used.
$GHNiMeXhVU70giNl$	the text between second and third $ sign is a salt used for hashing
vpm87wq/tB5X5rA8MYnsw8ssB7iyW.9gh5m/drfTmMJdvRtArB/3Xtyan1/DmOeBdpxs9cfKaDt0n15nqupvn/:	Hashed password

Let's generate the password for user 'kali'

Python 2.7.18 (default, Apr 20 2020, 20:30:41)

[GCC 9.3.0] on linux2

Type "help", "copyright", "credits" or "license" for more information.

>>> import crypt

>>> password="kali"

>>> hashing_scheme_with_salt="6GHNiMeXhVU70giNl$"

>>> crypt.crypt(password, hashing_scheme_with_salt)

'6GHNiMeXhVU70giNl$vpm87wq/tB5X5rA8MYnsw8ssB7iyW.9gh5m/drfTmMJdvRtArB/3Xtyan1/DmOeBdpxs9cfKaDt0n15nqupvn/'

>>>

A native python implementation

```
#!/usr/bin/env python

from __future__ import print_function
import struct
import io

try:
    range = xrange
except NameError:
    pass
```

```
def _left_rotate(n, b):
    """Left rotate a 32-bit integer n by b bits."""
    return ((n << b) | (n >> (32 - b))) & 0xffffffff


def _process_chunk(chunk, h0, h1, h2, h3, h4):
    """Process a chunk of data and return the new digest variables."""
    assert len(chunk) == 64

    w = [0] * 80

    # Break chunk into sixteen 4-byte big-endian words w[i]
    for i in range(16):
        w[i] = struct.unpack(b'>I', chunk[i * 4:i * 4 + 4])[0]

    # Extend the sixteen 4-byte words into eighty 4-byte words
    for i in range(16, 80):
        w[i] = _left_rotate(w[i - 3] ^ w[i - 8] ^ w[i - 14] ^ w[i - 16], 1)

    # Initialize hash value for this chunk
    a = h0
    b = h1
    c = h2
    d = h3
    e = h4

    for i in range(80):
        if 0 <= i <= 19:
            # Use alternative 1 for f from FIPS PB 180-1 to avoid bitwise not
            f = d ^ (b & (c ^ d))
            k = 0x5A827999
        elif 20 <= i <= 39:
            f = b ^ c ^ d
            k = 0x6ED9EBA1
        elif 40 <= i <= 59:
            f = (b & c) | (b & d) | (c & d)
            k = 0x8F1BBCDC
        elif 60 <= i <= 79:
            f = b ^ c ^ d
            k = 0xCA62C1D6

        a, b, c, d, e = ((_left_rotate(a, 5) + f + e + k + w[i]) & 0xffffffff,
                         a, _left_rotate(b, 30), c, d)

    # Add this chunk's hash to result so far
    h0 = (h0 + a) & 0xffffffff
    h1 = (h1 + b) & 0xffffffff
    h2 = (h2 + c) & 0xffffffff
    h3 = (h3 + d) & 0xffffffff
    h4 = (h4 + e) & 0xffffffff

    return h0, h1, h2, h3, h4


class Sha1Hash(object):
```

```
"""A class that mimics that hashlib api and implements the SHA-1 algorithm."""

name = 'python-sha1'
digest_size = 20
block_size = 64

def __init__(self):
    # Initial digest variables
    self._h = (
        0x67452301,
        0xEFCDAB89,
        0x98BADCFE,
        0x10325476,
        0xC3D2E1F0,
    )

    # bytes object with 0 <= len < 64 used to store the end of the message
    # if the message length is not congruent to 64
    self._unprocessed = b''
    # Length in bytes of all data that has been processed so far
    self._message_byte_length = 0

def update(self, arg):
    """Update the current digest.
    This may be called repeatedly, even after calling digest or hexdigest.
    Arguments:
        arg: bytes, bytearray, or BytesIO object to read from.
    """
    if isinstance(arg, (bytes, bytearray)):
        arg = io.BytesIO(arg)

    # Try to build a chunk out of the unprocessed data, if any
    chunk = self._unprocessed + arg.read(64 - len(self._unprocessed))

    # Read the rest of the data, 64 bytes at a time
    while len(chunk) == 64:
        self._h = _process_chunk(chunk, *self._h)
        self._message_byte_length += 64
        chunk = arg.read(64)

    self._unprocessed = chunk
    return self

def digest(self):
    """Produce the final hash value (big-endian) as a bytes object"""
    return b''.join(struct.pack(b'>I', h) for h in self._produce_digest())

def hexdigest(self):
    """Produce the final hash value (big-endian) as a hex string"""
    return '%08x%08x%08x%08x%08x' % self._produce_digest()

def _produce_digest(self):
    """Return finalized digest variables for the data processed so far."""
    # Pre-processing:
    message = self._unprocessed
    message_byte_length = self._message_byte_length + len(message)
```

```
        # append the bit '1' to the message
        message += b'\x80'

        # append 0 <= k < 512 bits '0', so that the resulting message length (in bytes)
        # is congruent to 56 (mod 64)
        message += b'\x00' * ((56 - (message_byte_length + 1) % 64) % 64)

        # append length of message (before pre-processing), in bits, as 64-bit big-endian
integer
        message_bit_length = message_byte_length * 8
        message += struct.pack(b'>Q', message_bit_length)

        # Process the final chunk
        # At this point, the length of the message is either 64 or 128 bytes.
        h = _process_chunk(message[:64], *self._h)
        if len(message) == 64:
            return h
        return _process_chunk(message[64:], *h)


def sha1(data):
    """SHA-1 Hashing Function
    A custom SHA-1 hashing function implemented entirely in Python.
    Arguments:
        data: A bytes or BytesIO object containing the input message to hash.
    Returns:
        A hex SHA-1 digest of the input message.
    """
    return Sha1Hash().update(data).hexdigest()


if __name__ == '__main__':
    # Imports required for command line parsing. No need for these elsewhere
    import argparse
    import sys
    import os

    # Parse the incoming arguments
    parser = argparse.ArgumentParser()
    parser.add_argument('input', nargs='*',
                        help='input file or message to hash')
    args = parser.parse_args()

    data = None
    if len(args.input) == 0:
        # No argument given, assume message comes from standard input
        try:
            # sys.stdin is opened in text mode, which can change line endings,
            # leading to incorrect results. Detach fixes this issue, but it's
            # new in Python 3.1
            data = sys.stdin.detach()

        except AttributeError:
            # Linux ans OSX both use \n line endings, so only windows is a
            # problem.
            if sys.platform == "win32":
                import msvcrt
```

```
            msvcrt.setmode(sys.stdin.fileno(), os.O_BINARY)
        data = sys.stdin

    # Output to console
    print('sha1-digest:', sha1(data))

else:
    # Loop through arguments list
    for argument in args.input:
        if (os.path.isfile(argument)):
            # An argument is given and it's a valid file. Read it
            data = open(argument, 'rb')

            # Show the final digest
            print('sha1-digest:', sha1(data))
        else:
            print("Error, could not find " + argument + " file." )
```

Digital signatures

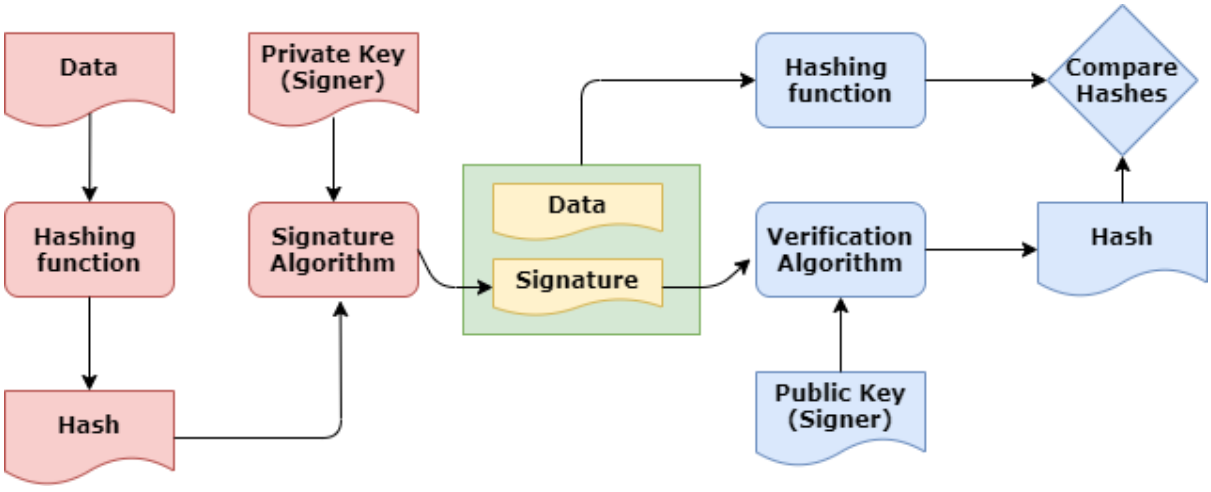
People use handwritten signatures to indicate authentication of their

- contracts (sales, insurance, employment, etc.)
- administrative papers (tax declarations, statements, etc.)
- transactions (banking)

In the digital word there is a need for a similar technique to identify a person, or any digital entity.

An electronic signature is defined as "data in electronic form which is attached to or logically associated with other data in electronic form and which is used by the signatory to sign" (eIDAS Article 3) [[8]]

Public key cryptography can be a solid base of digital signatures scheme:

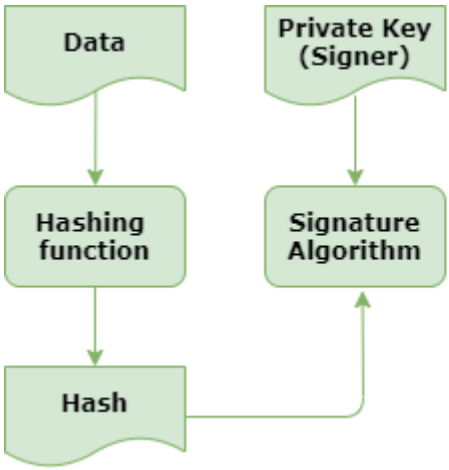


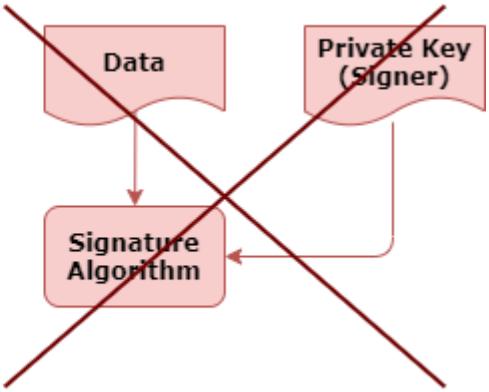
The workflow is as follows:

- each entity has a public + private key pair
- in general, those keys are different.
- private keys are used for signing, public keys are for verification

- sender generates the hash of data
- signature algorithm is applied to the hash value and private key
- signature is appended to the data
- the package being sent contains both the data and the signature
- receiver puts the data and signature to the verification algorithm which produces a hash
- if the senders hash and receivers hash is the same then the signature is valid

The digital signature is created by means of a secret private key. The original creator of the data can be identified.





RSA is commonly used as the signing algorithm. Signing large document would be time consuming. The hash of the document is much smaller, thus signing a hash is more efficient than signing the entire document.

Main usages of digital signatures:

- Authenticate messages – the private key is known by its oner so any valid digital signature can be created by only the sender who owns the key
- Check data integrity – if someone modifies the document then its hash will change so verification provide no matching hash. Receiver can verify that the document has no change since signed.

References

[1] Data Encryption Standard, Federal Information Processing Standard (FIPS) Publication 46, National Bureau of Standards, U.S. Department of Commerce, Washington D.C. (January 1977).

[2] https://sandilands.info/crypto/DataEncryptionStandard.html#x16-840008.5

[3] https://en.wikipedia.org/wiki/Data_Encryption_Standard

[4] https://www.youtube.com/watch?v=cVhlCzmb-v0

[5] R. Rivest: The MD5 Message-Digest Algorithm, 1992 https://www.ietf.org/rfc/rfc1321.txt

[6] https://github.com/timvandermeij/md5.py

[7] Rivest, Ronald L., Adi Shamir, and Leonard Adleman. "A method for obtaining digital signatures and public-key cryptosystems." Communications of the ACM 21.2 (1978): 120-126.

[8] REGULATION (EU) No 910/2014 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL of 23 July 2014 on electronic identification and trust services for electronic transactions in the internal market and repealing Directive 1999/93/EC
[9] Daemen, Joan, and Vincent Rijmen. "AES proposal: Rijndael." (1999). Federal Information Processing Standards Publication 197 Announcing the ADVANCED ENCRYPTION STANDARD (AES), November 26, 2001
[10] https://github.com/ajalt/python-sha1