Mid Term Exam (MTE) will be held in the middle of semester.
Please participate with your own computers with installed Octave and .m files.
During the MTE you must solve 2 problems:

1. Diffie-Hellman Key Agreement Protocol - DH KAP.
2. Man-in-the-Middle Attack (MiMA) for Diffie-Hellman Key Agreement Protocol - DH KAP.

The problems are presented in the site:
imimsociety.net
In section 'Cryptography':
Cryptography (imimsociety.net)

Please register to the site and after that you receive 10 Eur virtual money to purchase the problems.
For registration you should input the first 2 letters of your Surname and full Name, e.g. John Smith
Should register as **Sm John**.

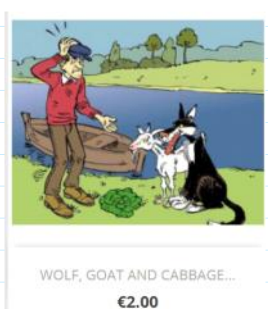
Please purchase the only one problem at a time.

If the solution is successful then you are invited to press the green button [Get reward].
No any other declaration about the solution results is required.
If the solution failed, then you must press the button [Return] in the top on the left side.

In the case of success 'Knowledge bank' will pay you the sum twice you have paid.
So, if the initial capital was 10 Eur of virtual money and you buy the problem of 2 Eur, then if the solution is correct your budget will increase up to 12 Eur.

You can solve the problems in imimsociety as many times as you wish to better prepare for MTE.

I advise you to try at first to solve the problem in 'Intellect' section to exercise the brains.
It is named as 'WOLF, GOAT AND CABBAGE TRANSFER ACROSS THE RIVER ALGORITHM'.
< https://imimsociety.net/en/home/15-wolf-goat-and-cabbage-transfer-across-the-river-algorithm.html>



Cryptography: information
confidentiality, integrity, authenticity, person identification

Symmetric cryptography -------------------- **Asymmetric cryptography** 1976

Asymmetric encryption

Symmetric encryption:

Symmetric encryption:

block ciphers

stream ciphers

H-functions, Message digest

HMAC H-Message Authentication Code

Asymmetric encryption

E-signature - Public Key Infrastructure - PKI

Blockchain, Cryptocurrecy, E-money

E-voting

Digital Rights Management - DRM (Marlin)

Etc.

A **cryptographic hash function** is a special class of hash function that has certain properties which make it suitable for use in cryptography. It is a mathematical algorithm that maps data of arbitrary finite size to a bit string of a fixed size (a hash function) which is designed to also be a one-way function, that is, a function which is infeasible to invert.

The only way to recreate the input data from an ideal cryptographic hash function's output is to attempt a brute-force search of possible inputs to see if they produce a match.

The input data is often called the *message*, and the output (the *hash value* or *hash*) is often called the *message digest* or simply the *digest*.

From <https://en.wikipedia.org/wiki/Cryptographic_hash_function>

M – finite length message

$h = H(M)$

$|h| = 256$ bits

$|h| = 28$ bits $= 7$ hexnumb.

$0000_b = 0_h \equiv 0_d$

$0001_b = 1_h \equiv 1_d$

$0010_b = 2_h \equiv 2_d$

$1001_b = 9_h \equiv 9_{10}$

$1010_b = A_h \equiv 10_{10}$

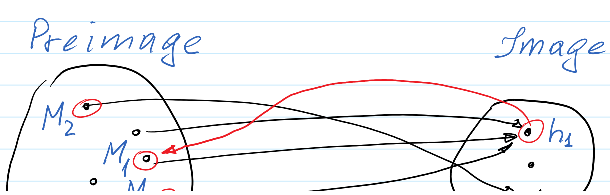
$1110_b = E_h \equiv 14_{10}$

$1111_b = F_h = 15_{10}$

Cryptographic hash functions have many information-security applications, notably in digital signatures, message authentication codes (HMACs), and other forms of authentication. They can also be used as ordinary hash functions, to index data in hash tables, for fingerprinting, to detect duplicate data or uniquely identify files, and as checksums to detect accidental data corruption. Indeed, in information-security contexts, cryptographic hash values are sometimes called (*digital*) *fingerprints*, ***message digest*** or just *hash values*, even though all these terms stand for more general functions with rather different properties and purposes.

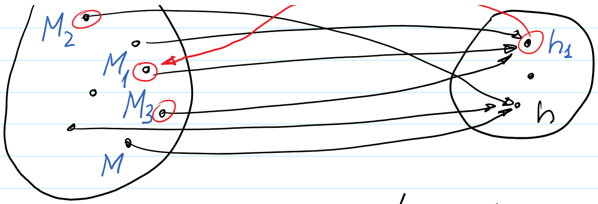
M – message ; $H(M) = h$

$M \in \{0,1\}^* ; \quad h \in \{0,1\}^{256} ; \quad H : \{0,1\}^* \to \{0,1\}^{256}$ //SHA256



1 GB → 256 bits

many – to – one



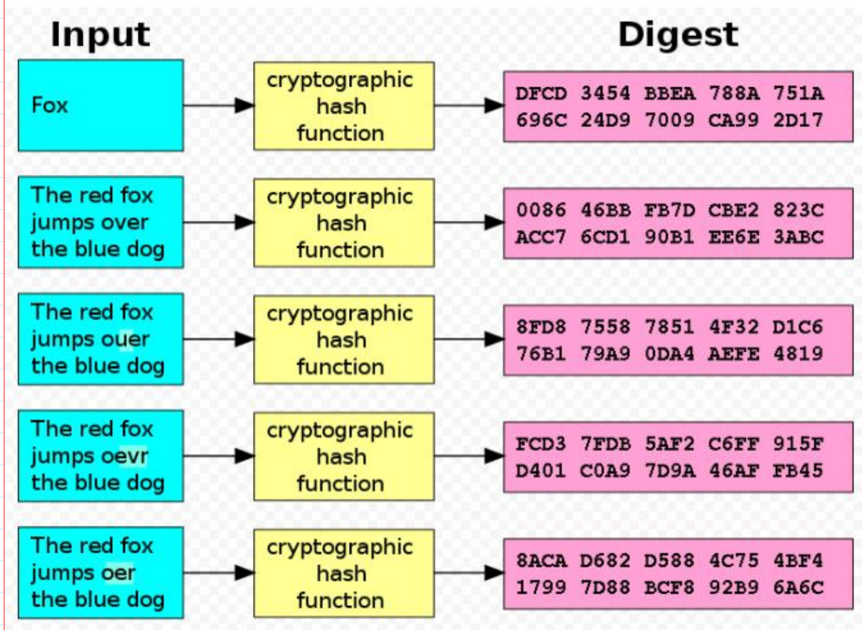
1 GB ⟶ 256 bits

many-to-one

$H(M_1) = H(M_2) = H(M_3) = \ldots = h_1$

For given h_1 it is infeasible to find any M_i satisfying:

$$\boldsymbol{H(M_i) = h_1}$$



40 Hex digits = 160 bits

SHA-1

SHA-1: $\{0,1\}^* \to \{0,1\}^{160}$

Avelange effect

2^{160} $\xrightarrow{\text{Birtdays}}$ 2^{80}

A cryptographic hash function (specifically SHA-1) at work. A small change in the input (in the word "over") drastically changes the output (digest). This is the so-called avalanche effect.

Properties

- It is quick to compute the hash value for any given finite message.
- A small change to a message should change the hash value so extensively that the new hash value appears uncorrelated with the old hash value.
- Security properties presented below.

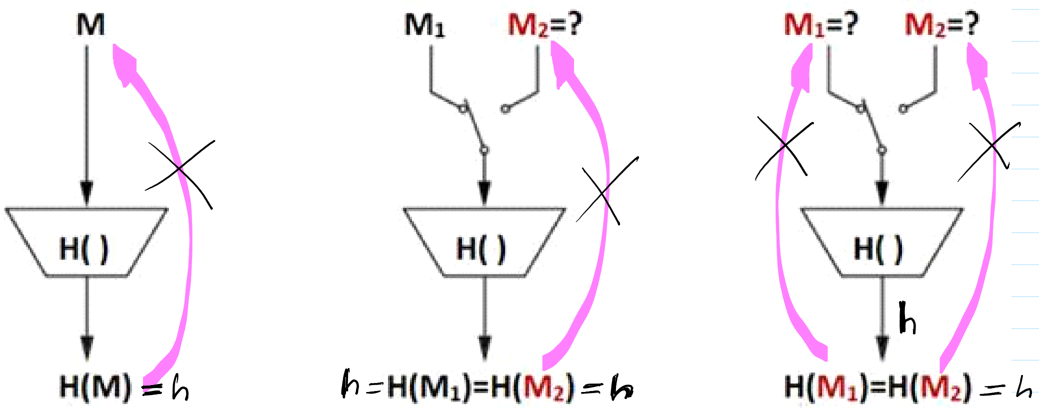
Most cryptographic hash functions are designed to take a string of any finite length as input and produce a fixed-length hash value.
A cryptographic hash function must be able to withstand all known types of cryptanalytic attack.
In theoretical cryptography, the security level of a cryptographic hash function has been defined using the following properties:

- ***Pre-image resistance***
  Given a hash value ***h*** it should be difficult to find any message ***M*** such that ***h*** = ***H(M)***. This concept is related to that of one-way function. Functions that lack this property are vulnerable to first preimage attacks.

- ***Second pre-image resistance***
  Given an input $\boldsymbol{M_1}$ it should be difficult to find (different) input $\boldsymbol{M_2}$ such that $\boldsymbol{H(M_1) = H(M_2)}$.

Functions that lack this property are vulnerable to second-preimage attacks.

- ***Collision resistance***
  It should be difficult to find any two different messages M_1 and M_2 such that $H(M_1) = M(M_2)$.
  Such a pair is called a cryptographic hash collision. This property is sometimes referred to as *strong collision resistance.* It requires a hash value at least twice as long as that required for preimage-resistance; otherwise **collisions** may be found by a birthday attack.[2]
  These properties form a hierarchy, in that collision resistance implies second pre-image resistance, which in turns implies pre-image resistance, while the converse is not true in general. [3]
  The weaker assumption is always preferred in theoretical cryptography, but in practice, a hash-functions which is only second pre-image resistant is considered insecure and is therefore not recommended for real applications.
  Informally, these properties mean that a malicious adversary cannot replace or modify the input data without changing its digest.
  Thus, if two strings have the same digest, one can be very confident that they are identical.



Preimage resistance **Second preimage resistance** **Collision resistance**

SHA-256 is collision resistant
For Bitcoin mining

Loan contract

$M_1 = 1000€$ loan. M_1 - is a valid loan contract.
$h = H(M_1)$; $|h| = 256$ bits

A: $PrK_A = x \longrightarrow Sign(PrK_A, h) = \sigma_h = (r_A, s_A)$

$M_1, \sigma_h, PuK_A \longrightarrow$ Jo: finds second preimage M_2 to create loan contract for 100 000 € such that $H(M_1) = H(M_2) = h$

Then the valid signature on M_1 is valid also on M_2.

A: $h = H(M_2) \longleftarrow M_2, \sigma_h$ claim to Alice to pay
Ver $(\sigma_h, h, PuK_A) = True$ 100 000 € (insted of 1000 €)

$A: \; h = H(M_2)$ ← M_2, σ_h — claim to Alice to pay 100 000 € (insted of 1000 €)

$Ver(\sigma_h, h, PuK_A) = True$

hd28.m - computing 28 bit length h-value in decimal form
h28.m - computing 28 bit length h-value in hexadecimal form
sha256.m - computing 256 bit length h-value in hexadecimal form

```
>> sha256('RootHash PrevHash 737327631')
ans = F4AE534CD226FAF7998C8424B348E020BA80639A687E93A0B8C5130EDC51E6DE
>> h28('RootHash PrevHash 737327631')
ans = C51E6DE
>> hd28('RootHash PrevHash 737327631')
ans = 206694110
>> dec2bin(ans)
ans = 1100010100011110011011011110
>> dec2hex(206694110)
ans = C51E6DE
```

Illustration nonce = 737327631 Consensus Mechanism in PoW mining

```
>> sha256('RootHash PrevHash 737327631')                      h28('....')
ans = F4AE534CD226FAF799 8C8424B348E020BA80639A687E93A0B8C5130ED C51E6DE
                                                                 C51E6DE

>> sha256('RootHash PrevHash 737327632')
ans = B856211DF2EE15E30AB770C1A43CE014ECFE573182AFD885B28D96854DBC5F21
>> sha256('RootHash PrevHash 737327633')
ans = 9C18C764E347A58E57AC3F7A3C2874D5889A0E802699FEA47EEFF8C03BFEDA69
>> sha256('RootHash PrevHash 737327634')
ans = 32B2108A70C39565485CCED9C948E5B7A0027D1EE98642E09D5E4D3D84E16814
>> sha256('RootHash PrevHash 737327635')
ans = A281AC77F5C9AEDEEFFDEDEA85DCEA1C5D76E4222AB80D8A456AEB2AA9EB0F44
```

$0_h \equiv 0000_2$; $F_h \equiv 1111_2$

h28('...') → 7 hex numb.
hd28('...') → decimal num.

Commitment

An illustration of the potential use of a cryptographic hash is as follows:
Alice poses a tough math problem to Bob and claims she has solved it.
Bob would like to try it himself, but would yet like to be sure that Alice is not bluffing.
Therefore, Alice writes down her solution, computes its hash and tells Bob the hash value (whilst keeping the solution secret).
Then, when Bob comes up with the solution himself a few days later, Alice can prove that she had the solution earlier by revealing it and having Bob hash it and check that it matches the hash value given to him before. (This is an example of a simple commitment scheme; in actual practice, Alice and Bob will often be

$P = NP$

$P \neq NP$

Elementary: Sherlock Holms and docto Watson computer programs, and the secret would be something less easily spoofed than a claimed puzzle solution).

Verifying the integrity of files or messages

Main article: File verification

An important application of secure hashes is verification of message integrity. Determining whether any changes have been made to a message (or a file), for example, can be accomplished by comparing message digests calculated before, and after, transmission (or any other event).

$$f \neq f'$$
$$H(f) \neq H(f')$$
$$h \neq h'$$
$$Sign(PrK_s, h) \neq Sign(PrK_s, h')$$

For this reason, most digital signature algorithms only confirm the authenticity of a hashed digest of the message to be "signed". Verifying the authenticity of a hashed digest of the message is considered proof that the message itself is authentic.

MD5, SHA1, or SHA2 hashes are sometimes posted along with files on websites or forums to allow verification of integrity.[6] This practice establishes a chain of trust so long as the hashes are posted on a site authenticated by HTTPS.

Password verification[edit]

Main article: password hashing

A related application is password verification (first invented by Roger Needham). Storing all user passwords as cleartext can result in a massive security breach if the password file is compromised. One way to reduce this danger is to only store the hash digest of each password. To authenticate a user, the password presented by the user is hashed and compared with the stored hash. (Note that this approach prevents the original passwords from being retrieved if forgotten or lost, and they have to be replaced with new ones.) **The password is often concatenated with a random, non-secret salt value** before the hash function is applied. The salt is stored with the password hash. Because users have different salts, it is not feasible to store tables of precomputed hash values for common passwords. Key stretching functions, such as PBKDF2, Bcrypt or Scrypt, typically use repeated invocations of a cryptographic hash to increase the time required to perform brute force attacks on stored password digests.

In 2013 a long-term Password Hashing Competition was announced to choose a new, standard algorithm for password hashing.

Proof-of-work

Main article: Proof-of-work system

A proof-of-work system (or protocol, or function) is an ***economic*** measure to deter denial of service attacks and other service abuses such as spam on a network by requiring some work from the service requester, usually meaning processing time by a computer. A key feature of these schemes is their asymmetry: the work must be moderately hard (but feasible) on the requester side but easy to check for the service provider.

One popular system — used in Bitcoin mining and Hashcash — **uses partial hash inversions to prove that work was done,** as a good-will token to send an e-mail. The sender is required to find a message whose hash value begins with a number of zero bits. The average work that sender needs to perform in order to find a valid message is

inversions to prove that work was done, as a good-will token to send an e-mail. The sender is required to find a message whose hash value begins with a number of zero bits. The average work that sender needs to perform in order to find a valid message is exponential in the number of zero bits required in the hash value, while the recipient can verify the validity of the message by executing a single hash function. For instance, in Hashcash, a sender is asked to generate a header whose 256 bit SHA-256 hash value has the first 18 bits as zeros. The sender will *on average* have to try $2^{4*18} = 2^{72}$ times to find a valid header.

$2^{20} = 1M$

File or data identifier

A message digest can also serve as a means of reliably identifying a file; several source code management systems, including Git, Mercurial and Monotone, use the sha1sum of various types of content (file content, directory trees, ancestry information, etc.) to uniquely identify them. Hashes are used to identify files on peer-to-peer filesharing networks.

Pseudorandom generation and key derivation

Hash functions can also be used in the generation of pseudorandom bits, or to derive new keys or passwords from a single secure key or password.

As of 2009, the two most commonly used cryptographic hash functions were MD5 and SHA-1. However, a successful attack on MD5 broke Transport Layer Security in 2008.

In February 2005, an attack on SHA-1 was reported that would find collision in about 2^{69} hashing operations, rather than the 2^{80} expected for a 160-bit hash function. In August 2005, another attack on SHA-1 was reported that would find collisions in 2^{63} operations. Though theoretical weaknesses of SHA-1 exist,[14][15] no collision (or near-collision) has yet been found. Nonetheless, it is often suggested that it may be practical to break within years, and that new applications can avoid these problems by using later members of the SHA family, such as SHA-2.

According to birthday paradox it is not required total scan all 2^{160} variants of message M. It is enough to scan $\sqrt{2^{160}} = 2^{80}$ variants.

SHA-2 (**Secure Hash Algorithm 2**) is a set of cryptographic hash functions designed by the United States National Security Agency (NSA).[3]

From <https://en.wikipedia.org/wiki/SHA-2>

SHA-2 includes significant changes from its predecessor, SHA-1.
The SHA-2 family consists of six hash functions with digests (hash values) that are 224, 256, 384 or 512 bits:

h28

SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256.

However, to ensure the long-term robustness of applications that use hash functions, there was a competition to design a replacement for SHA-2.
On October 2, 2012, Keccak was selected as the winner of the NIST hash function competition.
A version of this algorithm became a FIPS standard on August 5, 2015 under the name
SHA-3 <--> keccak-256 --> in Ethereum

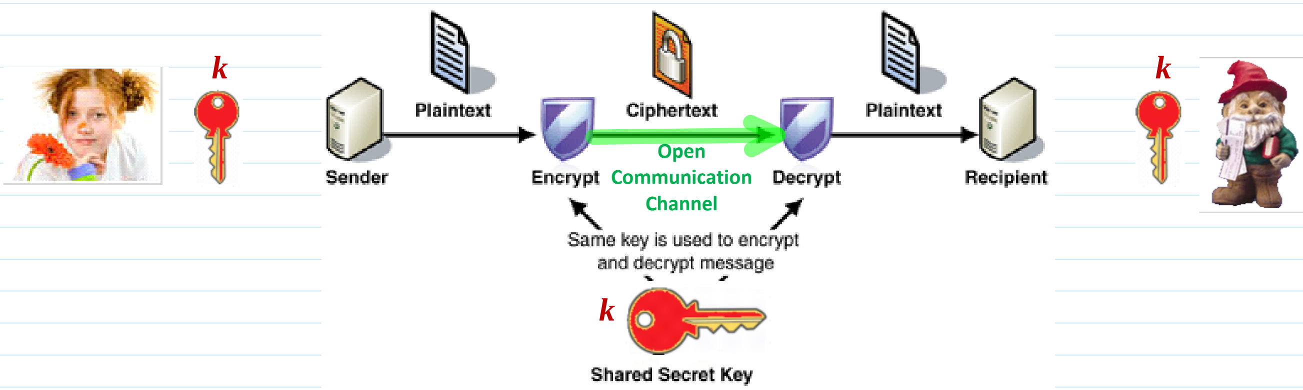
HMAC - H Message Authentication Code

Use in building other cryptographic primitives: symmetric e-signature realization
Hash functions can be used to build other cryptographic primitives.
For these other primitives to be cryptographically secure, care must be taken to build them correctly.
Message authentication codes (MACs) (also called keyed hash functions) are often built from hash functions. HMAC is such a MAC.

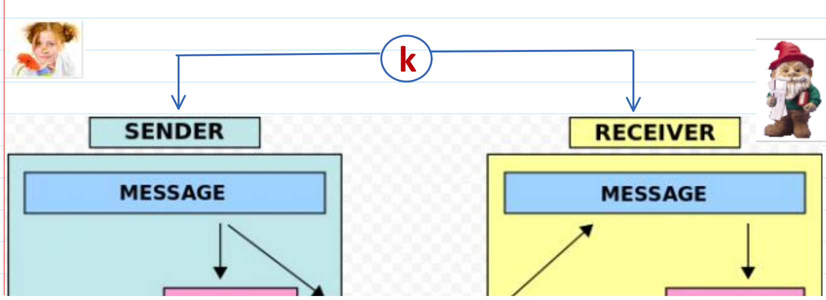
→ Information confidentiality
Authentication
Integrity

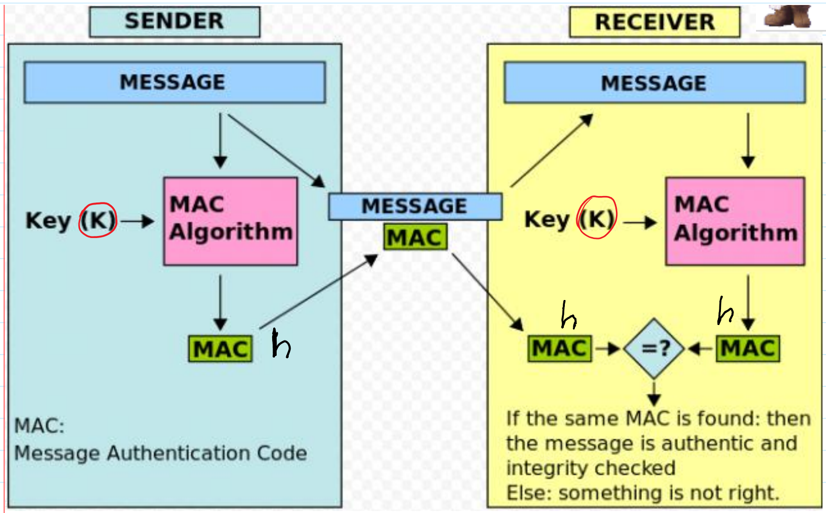
Keyed-hash message authentication code (**HMAC**) is a specific type of message authentication code (MAC) involving a cryptographic hash function (hence the 'H') in combination with a secret cryptographic key.
As with any MAC, it may be used to *simultaneously* verify both the *data integrity* and the *authentication* of a message.
Any cryptographic hash function, may be used in the calculation of an HMAC.
The cryptographic strength of the HMAC depends upon the cryptographic strength of the underlying hash function, the size of its hash output, and on the size and quality of the key.

Symmetric - Secret Key Encryption - Decryption



Integrity and authenticity by computing h-value and signing
HMAC based symmetric e-signature





Secret Key K

>> h=sha256('RootHash PrevHash 737327631 72652463582361747164677467 4781')

h = 0B398884242C678E83A363C689936815FEAD2AC97D1BE47FE0C7A04B95FAAA46

Angry Wich Zoe

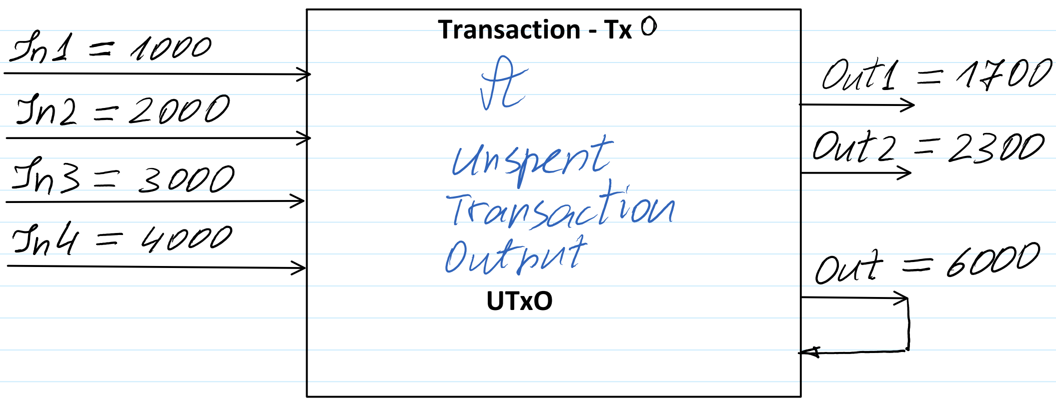
>> h=sha256('RootHash PrevHash 737327631 82846731775386352758237595 3978')

h = DFB648DAD49871F20D1C82E3D85BAF7E6EBF68066F02B27C40C6713AD97C77E0

Cryptocurrency transaction in Bitcoin
Using Unspent Transactions Output (UTxO) Paradigm

No.	Pajamos-Incomes	Išlaidos-Expenses	Likutis-Balance
In1.	Client1: 1000 Sat		1000 Sat
In2.	Client2: 2000 Sat	Out1. Firm 5: 1700 Sat	1300 Sat
In3.	Client3: 3000 Sat	Out2. Firm 6: 2300 Sat	2000 Sat
In4.	Client4: 4000 Sat	Out3. Firm 7:	6000 Sat
Total	**10 000 Sat**	**4000 Sat**	**6000 Sat**

Sum of Inputs =
= Sum of Outputs
Divisibility



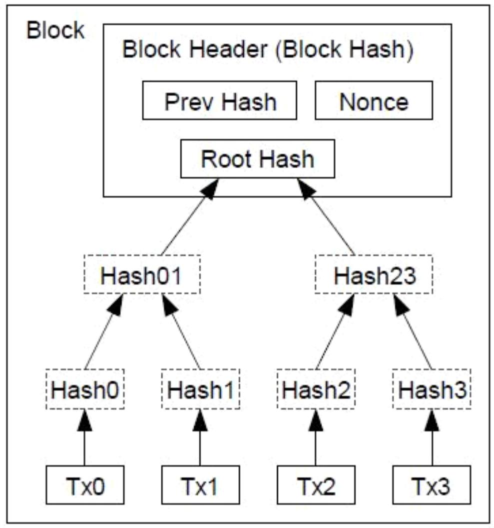
Bitcoin block structure

Antraštė

The Merkle Root of this hash tree is placed into the block's header along with the hash of the previous block (to be explained later) and a random number called a nonce (also to be explained later).
The block header will look something like this:

The block's header is then hashed with SHA256 producing an output that will serve as the block's identifier. Now having done all this can we go ahead and relay the block to the rest of the network? If you recall the last post, the answer is no. We still need to produce a valid proof of work.

From <https://chrispacia.wordpress.com/2013/09/02/bitcoin-mining-explained-like-youre-five-part-2-mechanics/>



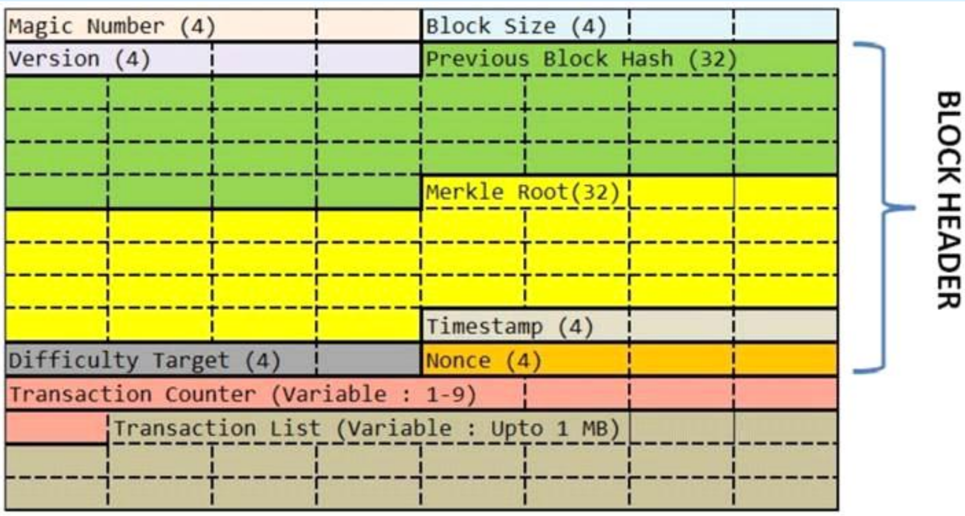
Hash0=H(Tx0)   Hash1=H(Tx1)   Hash2=H(Tx2)   Hash3=H(Tx3)

Hash01=H(Hash0||Hash1)   Hash23=H(Hash2||Hash3)

RootHash=Hash(Hash01||Hash23)

32B2108A70C39565485CCED9C948E5B7A0027D1EE98642E09D5E4D3D84E16814   A281AC77F5C9AEDEEFFDEDEA85DCEA1C5D76E4222AB80D8A456AEB2AA9EB0F44

856211DF2EE15E30AB770C1A43CE014ECFE573182AFD885B28D96854DBC5F21



$B = Inf \,||\, nonce$

$H(B) = h_M$

$nonce_M \rightarrow$

$H(B) = h \stackrel{?}{=} h_M$

https://bitcoin.org/en/glossary/block-header

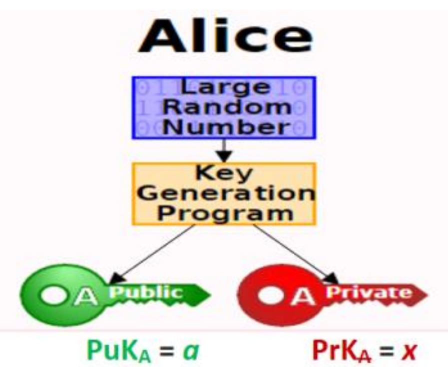
An 80-byte header belonging to a single block which is hashed repeatedly to create proof of work.

Structure of a Bitcoin Transaction Blockheader.

Field	Purpose	Updated when...	Size (Bytes)
Version	Block version number	When software upgraded	4
hashPrevBlock	256-bit hash of the previous block header	A new block comes in	32
hashMerkleRoot	256-bit hash based on all of the transactions in the block	A transaction is accepted	32
Time	Current timestamp as seconds since 1970-01-01T00:00 UTC	Every few seconds	4
Bits	Current target in compact format	The difficulty is adjusted	4
Nonce	32-bit number (starts at 0)	A hash is tried (increments)	4

Till this place

Public Key or Asymmetric Cryptography (PKC)

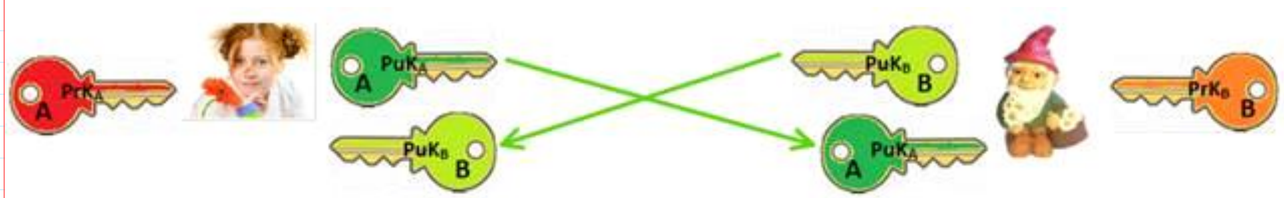


PP = (p, g).
Strong prime number p in real cryptography is of order : $p \sim 2^{2048}$
Strong prime number p in our examples is of order: $p \sim 2^{28}$
>> p=genstrongprime(28)

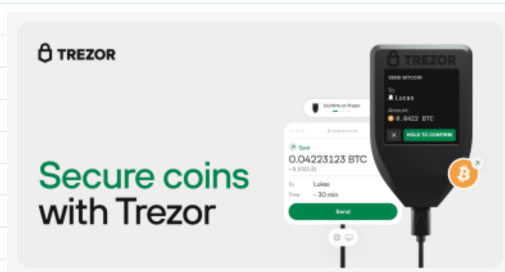
Key generation

- Randomly choose a private key x with $1 < x < p - 1$.
- The private key is $PrK = x$ = int64(**randi(**p**-1)**)

  Compute $a = g^x \bmod p$.
- The public key is $PuK = a = g^x \bmod p$.



[Trezor Hardware Wallet (Official) | Bitcoin & Crypto Security](trezor.io)
The safest cold storage wallets for crypt security and financial independence. Easily use, store, and protect Bitcoins.
trezor.io



C:\Users\Eligijus\Documents\500 SOFTAS App 2023\Python 3.9.1\111.ECDSA-secp256k1_Python.Edv.Rep\ECC

C:\Users\Eligijus\AppData\Local\Programs\Python\Launcher\py.exe

```
ECCDS python app
Please input required command:
        1  - Generate new ECC private and public keys
        2  - Export private and public keys
        3  - Export private key
        4  - Export public key
        5  - Load private key
        6  - Load data file
        7  - Sign loaded file
        8  - Load public key
        9  - Verify signature
        10 - Export signature
        11 - Load signature
        12 - Draw secp256k1 graph in real numbers
        13 - Draw secp256k1 graph over finite field
        exit/e - Exit app
Input command:
```

Confidentiality, Integrity and Authenticity by encryption, computing h-value and signing

$\mathcal{A}$: message M to be sent to $\mathcal{B}$.

1. Parties agree on the common secret key k.
2. $\mathcal{A}$ encrypts message using symmetric encryption algorithm, e.g. AES128: $C = AES(k, M)$; $|C| \approx |M|$.
3. The HMAC-value on C is computed: $h = HMAC(k, C)$

$\mathcal{A}$: —— C, h ——→ $\mathcal{B}$:

1. Computes HMAC-value on C
   $h' = HMAC(k, C)$
2. Verifies received h-value h with computed h'
   $Ver(\text{If } h = h') = \{T, F\}$
3. Decrypts C by k
   $D(k, C) = M$.

In the case of Asymmetric cryptography:

Confidentiality Integrity and Authenticity is realized by encryption, computing h-value and signing

A: message M to be sent to B.

1. Parties agree on the common secret key k.
2. A encrypts message using symmetric encryption algorithm, e.g. AES128: $C = AES(k, M)$; $|C| \approx |M|$.
3. The h-value of C is computed: $h = sha256(C)$
4. The signature is placed on h : $Sign(PrK_A, h) = \sigma = (r, s)$

A: $PrK_A = x$; $PuK_A = a$.

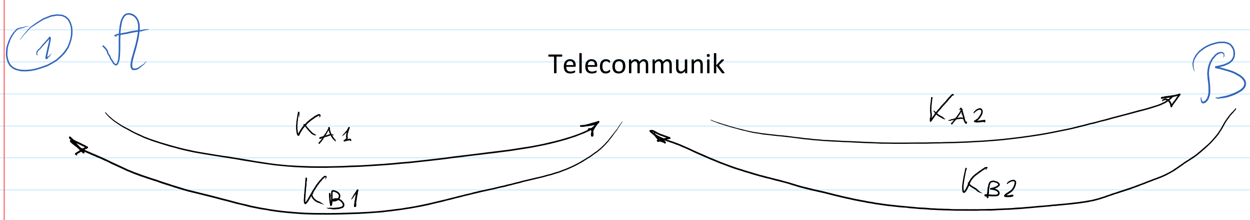
$C, \sigma = (r, s)$ →

B: $PrK_B = y$; $PuK_B = b$.

$PuK_A = a$.

1. Computes h-value of C
   $h = sha256(C)$
2. Verifies signature on h
   $Ver(PuK_A, \sigma, h) = \{T, F\}$
3. Decrypts C by k
   $D(k, C) = M$.

(1)



$$K_{AB} = \underbrace{(K_B)^x \bmod p}_{K_1} = K = \underbrace{(K_A)^y \bmod p}_{K_2} = K_{BA}$$

m – message

$Enc(K_1, m) = C_1$ →

$Dec(K_1, C_1) = m$

↓

Data center

$Enc(K_2, m) = C_2$ →

$Enc(K_2, m) = C_2 \longrightarrow Dec(K_2, C_2) = m$

(2) Telecommunik — software and its updating

Backdoors →

```
>> sha256('RootHash PrevHash 737327631')
ans = F4AE534CD226FAF7998C8424B348E020BA80639A687E93A0B8C5130EDC51E6DE
>> h28('RootHash PrevHash 737327631')
ans = C51E6DE
>> hd28('RootHash PrevHash 737327631')
ans = 206694110
>> dec2bin(ans)
ans = 1100010100011110011011011110
>> dec2hex(206694110)
ans = C51E6DE
```

Hash functions based on block ciphers

There are several methods to use a block cipher to build a cryptographic hash function, specifically a one-way compression function.
The methods resemble the block cipher modes of operation usually used for encryption.
Many well-known hash functions, including MD4, MD5, SHA-1 and SHA-2 are built from block-cipher-like components

HMAC can be constructed form the block cipher using cipher block chaining (CBC) mode of operation.

AES_{k}_CBC

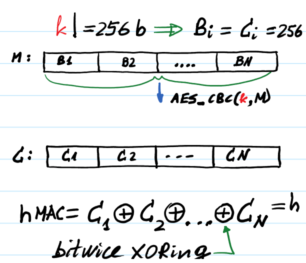
M – to be signed.

$C = AES\_CBC(k, M) \rightarrow 256$

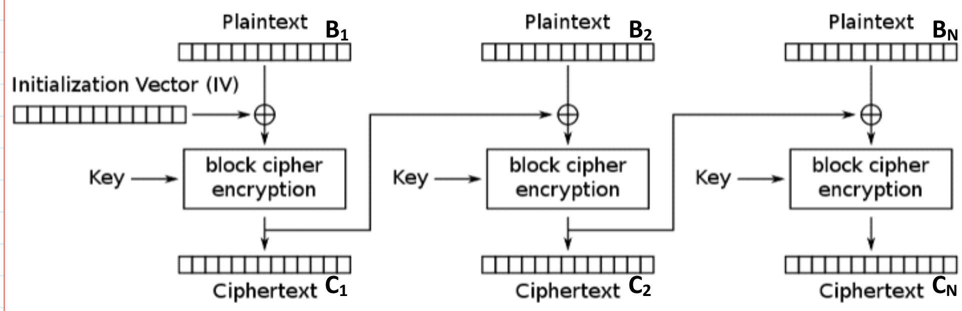
CBC-MAC

Cipher block chaining message authentication code (**CBC-MAC**) is a technique for constructing a message authentication code from a block cipher. The message is encrypted with some block cipher algorithm in CBC mode to create a chain of blocks such that each block depends on the proper encryption of the previous block.
This interdependence ensures that a change to any of the plaintext bits will cause the final encrypted block to change in a way that cannot be predicted or counteracted without knowing the key to the block cipher.

From <https://en.wikipedia.org/wiki/CBC-MAC>



Plaintext Plaintext Plaintext



Cipher Block Chaining (CBC) mode encryption

$hMAC = C_1 \oplus C_2 \oplus \ldots \oplus C_N = h$

bitwice XORing

$|h| = 256$ b.

Chosen Plaintext Attack

A:

$E_{CBC}(k, m) = c$ } encrypt & hash

$H_{CBC}(k, c) = h$

c, h →

B: 1) $H_{CBC}(k, c) = h'$

? $h \stackrel{?}{=} h'$ if ok

2) $D(k, c) = m$