

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

The problems are presented in the site:

<https://imimsociety.net/en/14-cryptography>

In section 'Cryptography':

[Cryptography \(imimsociety.net\)](https://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.

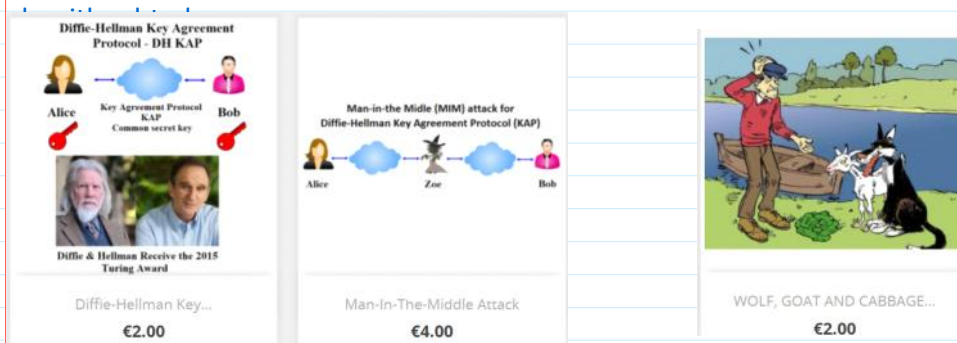
So, if the initial capital was 10 Eur of virtual money and you buy the problem of 2 Eur, then if the solution is correct your budget will increase up to 12 Eur.

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->



Cryptography: information
confidentiality, integrity, authenticity, person identification

Symmetric cryptography ----- Asymmetric cryptography 1976

Symmetric encryption:

block ciphers
stream ciphers

H-functions, Message digest

HMAC H-Message Authentication Code

Asymmetric encryption

E-signature - Public Key Infrastructure - PKI

Blockchain, Cryptocurrency, 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

$$h = H(M)$$

$$|h| = 256 \text{ bits}$$

$$|h| = 28 \text{ bytes}$$

$$= 7 \text{ hex num.}$$

$$0000_b = 0_h \equiv 0_d$$

$$0001_b = 1_h \equiv 1_d$$

$$0010_b = 2_h \equiv 2_d$$

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

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

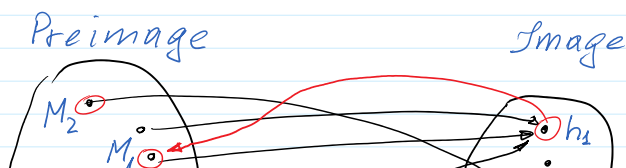
$$1110_b = E_h \equiv 14_{10}$$

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

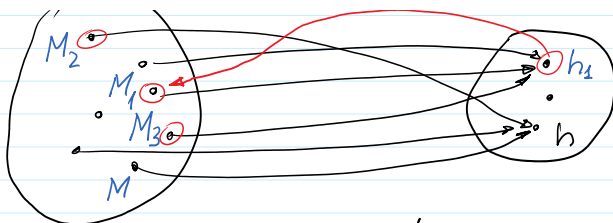
Cryptographic hash functions have many [information-security](#) applications, notably in [digital signatures](#), [message authentication codes](#) (HMACs), and other forms of [authentication](#). They can also be used as ordinary [hash functions](#), to index data in [hash tables](#), for [fingerprinting](#), 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.

$$M - \text{message}; H(M) = h$$

$$M \in \{0,1\}^*; h \in \{0,1\}^{256}; H: \{0,1\}^* \rightarrow \{0,1\}^{256} \parallel \text{SHA256}$$



$$1 \text{ GB} \rightarrow 256 \text{ bits}$$



1 GB $\rightarrow$ 256 bits

many-to-one

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

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

$$H(M_i) = h_1$$

Input		Digest
Fox	cryptographic hash function	DFCD 3454 BBEA 788A 751A 696C 24D9 7009 CA99 2D17
The red fox jumps over the blue dog	cryptographic hash function	0086 46BB FB7D CBE2 823C ACC7 6CD1 90B1 EE6E 3ABC
The red fox jumps over the blue dog	cryptographic hash function	8FD8 7558 7851 4F32 D1C6 76B1 79A9 0DA4 AEFE 4819
The red fox jumps over the blue dog	cryptographic hash function	FCD3 7FDB 5AF2 C6FF 915F D401 C0A9 7D9A 46AF FB45
The red fox jumps over the blue dog	cryptographic hash function	8ACA D682 D588 4C75 4BF4 1799 7D88 BCF8 92B9 6A6C

40 Hex digits = 160 bits

SHA-1

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

Avalanche effect

$$2^{160} \xrightarrow[\text{Part.}]{\text{Birthday}} 2^{80}$$

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** (as opposite with Check Redundancy Code - CRC) 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

to that of [one-way function](#). Functions that lack this property are vulnerable to [first preimage attacks](#).

- **Second pre-image resistance**

Given an input M_1 it should be difficult to find (different) input M_2 such that $H(M_1) = H(M_2)$. Functions that lack this property are vulnerable to [second-preimage attacks](#).

- **Collision resistance**

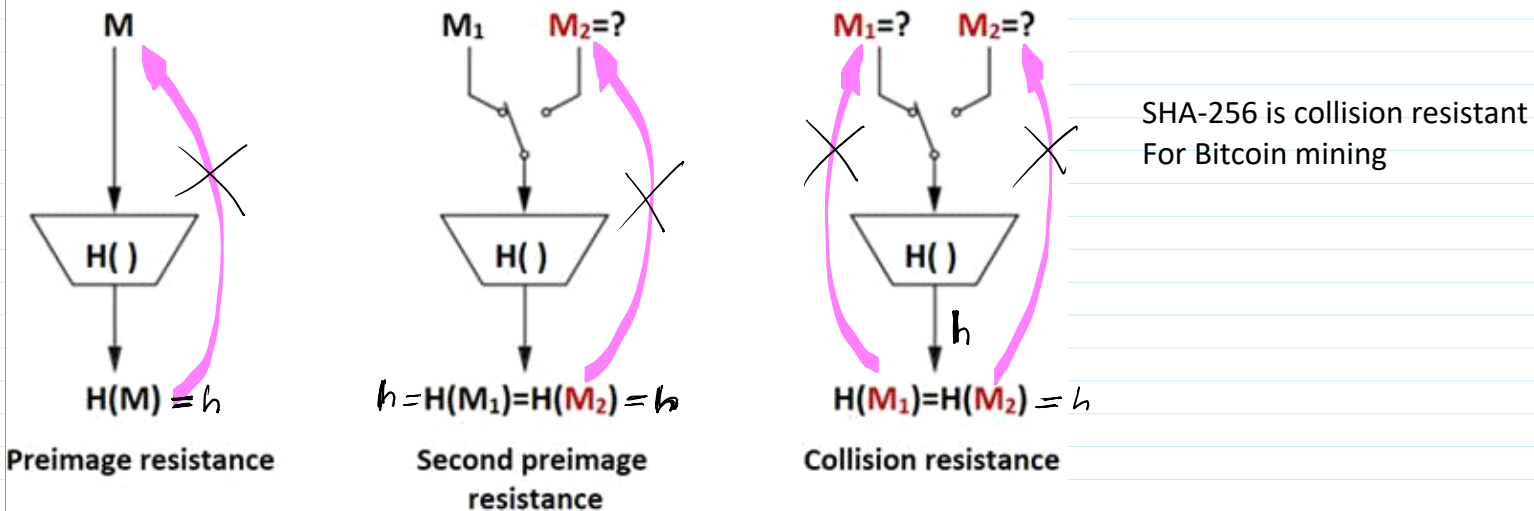
It should be difficult to find **any** two different messages M_1 and M_2 such that $H(M_1) = H(M_2)$. Such a pair is called a cryptographic [hash collision](#). This property is sometimes referred to as *strong collision resistance*. It requires a hash value at least twice as long as that required for 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.



Loan contract

$M_1 = 1000 \text{€ loan.}$

M_1 - is a valid loan contract.

$h = H(M_1); |h| = 256 \text{ bits}$

$A: PK_A = x \rightarrow \text{Sign}(PK_A, h) = \sigma_h = (r_A, s_A)$

$M_1, \sigma_h, PK_A \rightarrow$

I_o : finds second preimage M_2 to create loan contract for 100000€ such that $H(M_1) = H(M_2) = h$

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

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

A: $h = H(M_2)$ ← $M_2, \tilde{G}_h$ claim to Alice to pay
 Ver $(\tilde{G}_h, h, \text{PK}_A) = \text{True}$ 100 000 € (instead of 1000€)

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

```
>> h=hd28('Hello Bob')
```

```
h = 198770750
```

```
>> dec2bin(h)
```

```
ans = 1011110110010000000000111110
```

```
>> h=h28('Hello Bob')
```

```
h = BD9003E
```

```
>> hex2bin(h)
```

```
ans = 1011110110010000000000111110
```

Cryptography, Internet of Things (IoT), Blockchain, Cryptocurrency Business Value Chain



Producer



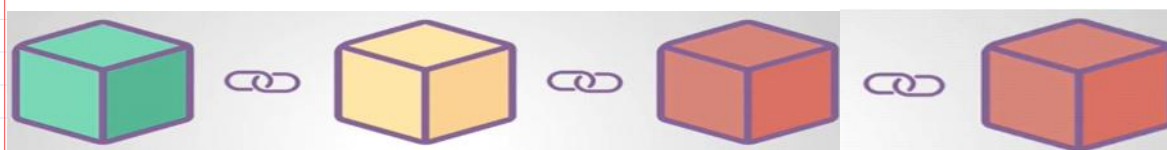
IoT



Wholesaler

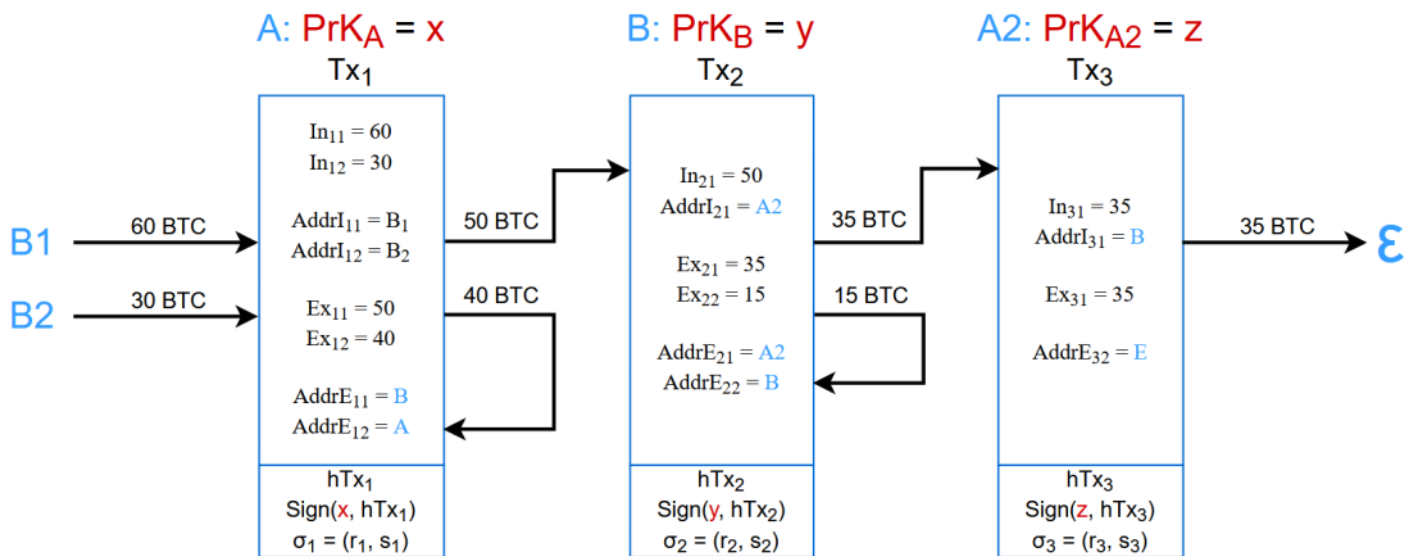
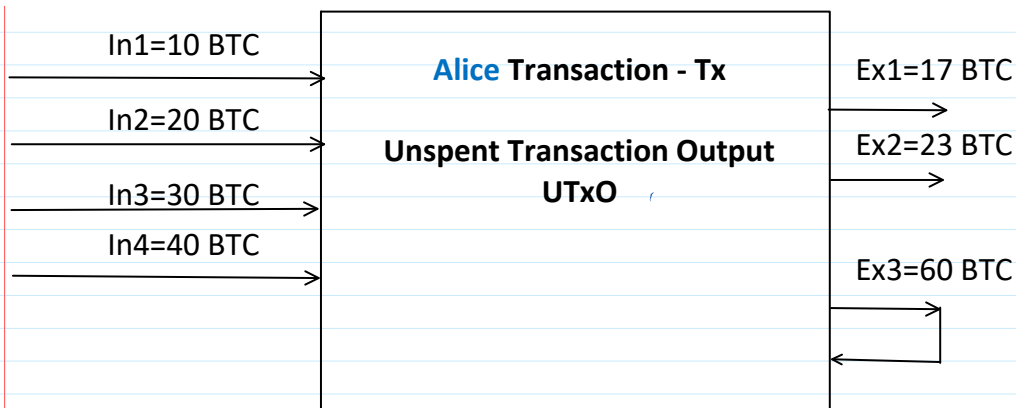


Retailer



Blockchain

Transactions and Tokens in Ethereum blockchain are controlled by Smart Contracts. Smart Contracts are controlled by *Events* that are initialized by Oracles created by IoT. Smart Contracts after receiving Oracles initiates Transactions.



Magic Number (4)	Block Size (4)
Version (4)	Previous Block Hash (32)
	Merkle Root(32)
	Timestamp (4)
Difficulty Target (4)	Nonce (4)
Transaction Counter (Variable : 1-9)	
Transaction List (Variable : Upto 1 MB)	

BLOCK HEADER

←

```
>> 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

Let Difficulty Target is 18 leading zeroes in h-value.

```
>> sha256('RootHash PrevHash 737327631')
ans = F4AE534CD226FAF7998C8424B348E020BA80639A687E93A0B8C5130EDC51E6DE
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
```

The probability to get 1 leading zero in h-value is $\Pr(1 \text{ leading zero}) = 1/16$
In average to get 1 leading zero we need to perform 16 h-value computations.

The probability to get 18 leading zero in h-value is $\Pr(18 \text{ leading zeroes}) = (1/16)^{18}$
In average to get 18 leading zero we need to perform 9 223 372 036 854 775 807 h-value computations.

```
>> int64(16^18)
ans = 9223372036854775807
```