Mid Term Exam (MTE) will be held in 30 of March at 13:30 contactly in 506 a. during a lecture.
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.

Then '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.

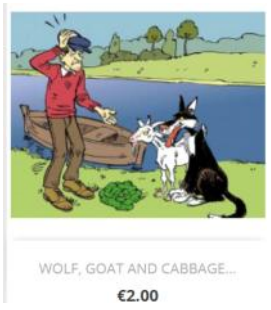
You can solve the problems in imimsociety as many times as you wish to better prepare for MTE.

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>

The questions concerning the MTE you can ask at the end of the lectures.

Diffie-Hellman Key Agreement Protocol - DH KAP
Alice
Key Agreement Protocol KAP Common secret key
Bob
Diffie & Hellman Receive the 2015 Turing Award
Diffie-Hellman Key...
€2.00

Man-in-the Midle (MIM) attack for Diffie-Hellman Key Agreement Protocol (KAP)
Alice
Zoe
Bob
Man-In-The-Middle Attack
€4.00



Principles of Public Key Cryptography

Instead of using single symmetric key shared in advance by the parties for realization of symmetric cryptography, asymmetric cryptography uses two ***mathematically*** related keys named as private key and public key we denote by **PrK** and **PuK** respectively.
PrK is a secret key owned ***personally*** by every user of cryptosystem and must be kept secretly. Due to the great importance of **PrK** secrecy for information security we labeled it in red color. **PuK** is a non-secret

personal key and it is known for every user of cryptosystem and therefore we labeled it by green color. The loss of **PrK** causes a dramatic consequences comparable with those as losing password or pin code. This means that cryptographic identity of the user is lost. Then, for example, if user has no copy of **PrK** he get no access to his bank account. Moreover his cryptocurrencies are lost forever. If **PrK** is got into the wrong hands, e.g. into adversary hands, then it reveals a way to impersonate the user. Since user's **PuK** is known for everybody then adversary knows his key pair (**PrK**, **Puk**) and can forge his Digital Signature, decrypt messages, get access to the data available to the user (bank account or cryptocurrency account) and etc.
Let function relating key pair (**PrK**, **Puk**) be ***F***. Then in most cases of our study (if not declared opposite) this relation is expressed in the following way:

PuK=***F***(**PrK**): **PrK = *x* = randi(*p*-1);** **PuK = *a* = *g*^*x* mod *p*.**

In open cryptography according to Kerchoff principle function ***F*** must be known to all users of cryptosystem while security is achieved by secrecy of cryptographic keys. To be more precise to compute **PuK** using function ***F*** it must be defined using some parameters named as public parameters we denote by **PP** and color in blue that should be defined at the first step of cryptosystem creation. Since we will start from the cryptosystems based on discrete exponent function then these public parameters are

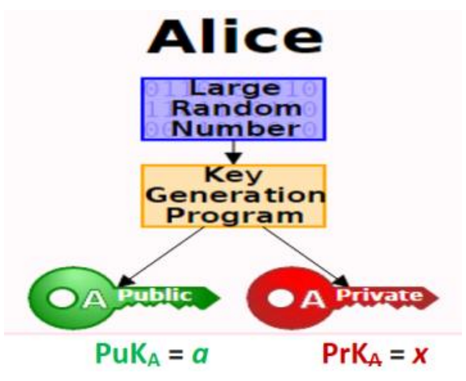
PP = (***p***, ***g***).

Notice that relation represents very important cause and consequence relation we name as the direct relation: when given **PrK** we compute **PuK**.
Let us imagine that for given ***F*** we can find the inverse relation to compute **PrK** when **PuK** is given. Abstractly this relation can be represented by the inverse function $\boldsymbol{F}^{-1}$. Then

PrK=$\boldsymbol{F}^{-1}$(**PuK**).

In this case the secrecy of **PrK** is lost with all negative consequences above. To avoid these undesirable consequences function ***F*** must be **one-way function** – **OWF**. In this case informally OWF is defined in the following way:
1. The computation of its direct value **PuK** when **PrK** and ***F*** in are given is effective.
2. The computation of its inverse value **PrK** when **PuK** and ***F*** are given is infeasible, meaning that to find $\boldsymbol{F}^{-1}$ is infeasible.

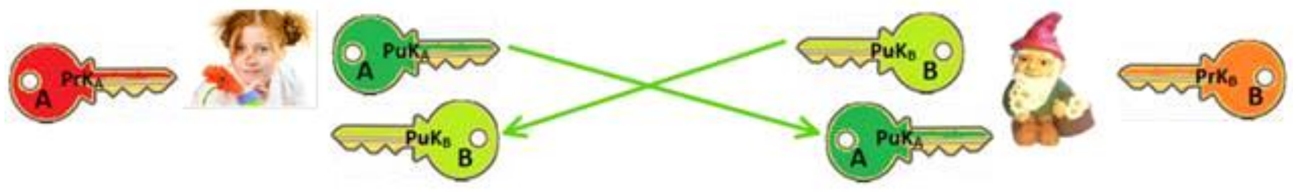
The one-wayness of ***F*** allow us to relate person with his/her **PrK** through the **PuK**. If ***F*** is 1-to-1, then the pair (**PrK**, **Puk**) is unique. So **PrK** could be reckoned as a unique secret parameter associated with certain person. This person can declare the possession or **PrK** by sharing his/her **PuK** as his public parameter related with **PrK** and and at the same time not revealing **PrK**.
So, every user in asymmetric cryptography possesses key pair (**PrK**, **PuK**). Therefore, cryptosystems based on asymmetric cryptography are named as **Public Key CryptoSystems** (PKCS).
We will consider the same two traditional (canonical) actors in our study, namely Alice and Bob.
Everybody is having the corresponding key pair ($\mathbf{PrK_A}$, $\mathbf{PuK_A}$) and ($\mathbf{PrK_B}$, $\mathbf{PuK_B}$) and are exchanging with their public keys using open communication channel as indicated in figure below.



Public Parameters **PP** = (***p***, ***g***).
Strong prime number ***p*** in real cryptography is of order : ***p*** ~ $\mathbf{2^{2048}}$
Strong prime number ***p*** in our examples is of order: ***p*** ~ $\mathbf{2^{28}}$
>> p=genstrongprime(28)
Key generation

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

  Compute ***a* = *g*^*x* mod *p***.
- The public key is **PuK = *a* = *g*^*x* mod *p***.



Security: PrK compromization: for given a, p, g find PrK$=x$ from the equation $a = g^x \bmod p \quad / \mathrm{dlog}_g$

$$\mathrm{dlog}_g a = \mathrm{dlog}_g (g^x \bmod p)$$

$$x \cdot (\mathrm{dlog}_g g \bmod p) = \mathrm{dlog}_g a$$

$$x \cdot 1 = \mathrm{dlog}_g a$$

$$\underline{x = \mathrm{dlog}_g a}$$

Discrete Logarithm Problem (DLP)

1. Criteria: parameters (p, g) must be chosen in such a way that DLP must be infeasible.

But there exist such groups where DLP is feasible.

Secure prime number p must be 2048 bit length, i.e., p =~ 2^{2048} ~ 10^{700}.

Asymmetric Signing - Verification

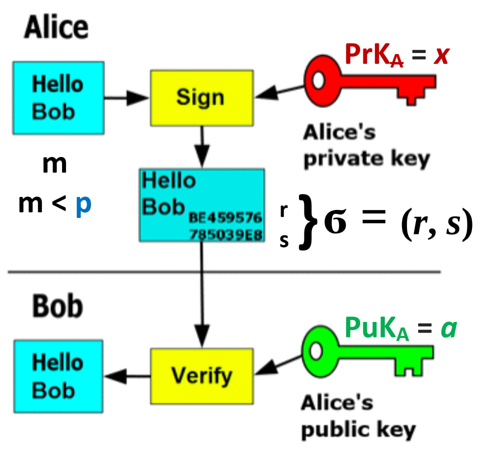
ϭ=Sign(PrK$_A$, m) = (r, s)

V=Ver(PuK$_A$, m, σ), V$\in$\{True, False\} $\equiv$ \{1, 0\}

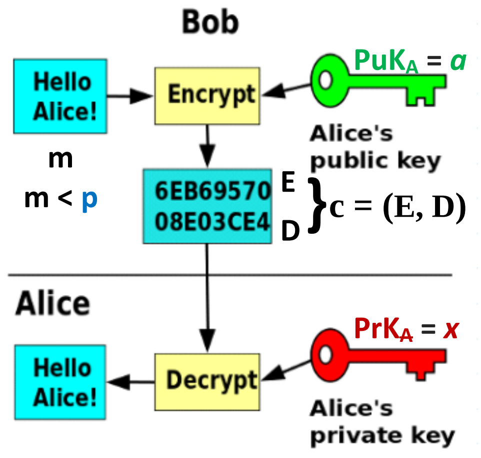
Asymmetric Encryption - Decryption

c=Enc(PuK$_A$, m) = (ε, δ) =(E, D)

m=Dec(PrK$_A$, c)



Authenticity



ε

δ

{

Confidentiality

1. ElGamal Cryptosystem

1.Public Parameters generation **PP** = (p, g).

Generate strong prime number **p**: >> p=genstrongprime(28) % strong prime of 28 bit length

Strong prime number p: defines the set Z_p*= {1, 2, 3, …, p-1}, where multiplication operations **mod p** are defined. Z_p*is an algebraic group where division operations are defined as well.

Find a generator **g** in Z_p*= {1, 2, 3, …, p-1} using condition:

Strong prime p=2q+1, where q is prime, then g is a generator of Z_P* iff

$g^q \neq 1$ mod p and $g^2 \neq 1$ mod p.

Declare **Public Parameters** to the network **PP** = (p, g); **p = 268435019**; **g=2;**

2^28-1= 268,435,455

```
>> 2^28-1
ans = 2.6844e+08
>> int64(2^28-1)
ans = 268435455
```

PrK = x <-- randi ==> PuK = a = g^x mod p

Compatibility relations of modular arithmetic:

(a + b) mod p = (a mod p + b mod p) mod p. >> mod(a+b,p)

(a * b) mod p =((a mod p) * (b mod p)) mod p. >> mod(a*b,p)

a^e mod p = (a mod p)^e mod p. **>> mod_exp(a,e,p)**

2. El-Gamal E-Signature

The **ElGamal signature scheme** is a digital signature scheme which is based on the difficulty of computing discrete logarithms.

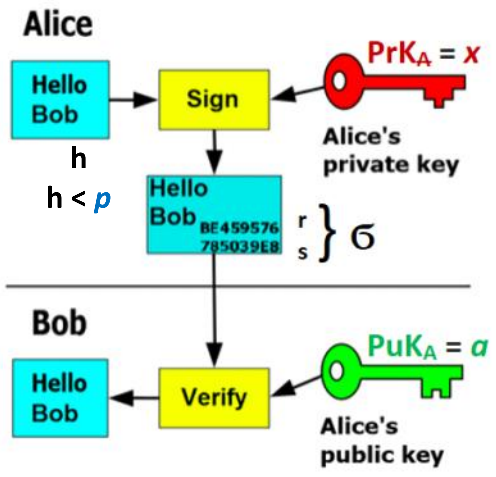
It was invented by Taher ElGamal in 1984. The ElGamal signature algorithm is rarely used in practice.

A variant developed at NSA and known as the Digital Signature Algorithm is much more widely used.

The ElGamal signature scheme allows a third-party to confirm the authenticity of a message sent over an insecure channel.

EC Gamal sign. $\longrightarrow$ Digital Signature Alg. (DSA) NSA

$\longrightarrow$ Elliptic Curve DSA - ECDSA Certicom



Authenticity

Signature creation for message M >> p.

1. Compute decimal h-value h=H(M); h<p: SHA-256.
2. Generate >> i =int64(randi(p-1)) % such that **gcd**(i, p-1)=**1**.
3. Compute i^{-1} mod (p-1). You can use the function

```
 >> gcd(i, p-1)
ans = 1
 >> i_m1=mulinv(i, p-1);
```

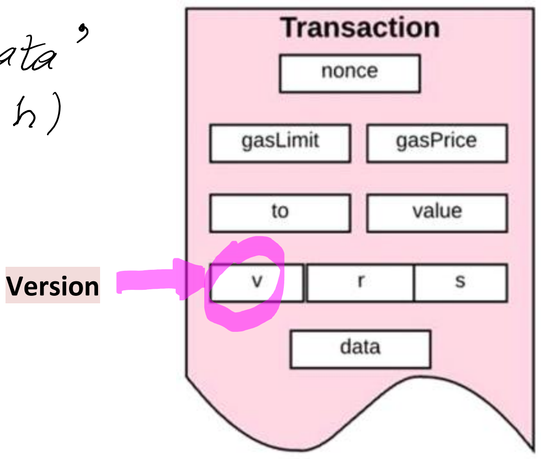
1. Compute r = g^i **mod** p.
2. Compute s = (h - xr)i^{-1} mod (p-**1**).
3. Signature on h-value h is σ = (r, s)

   Sign(x, h) = σ = (r, s).

$Tx =$ ' nonce || gasLimit || gasPrice || to || value || data '

$h = H(Tx) \longrightarrow \sigma = (r, s) = Sign(PrK, h)$

GasLimit is evaluated by Wey.
1 Wey = 10^{-18} Eth --> 1 Eth = 10^{18} Wey.
Usuausally 1Gas is thousands of Wey's.



1.***Signature creation***
To sign any finite message ***M*** the signer performs the following steps using public parametres **PP**.

- Compute **h=H(*M*)**. E.g. **SHA-256.**
- Choose a random ***i*** such that $\mathbf{1} < \boldsymbol{i} < \boldsymbol{p} - \mathbf{1}$ and $\mathbf{gcd}(\boldsymbol{i}, \boldsymbol{p} - \mathbf{1}) = \mathbf{1}$.
- Compute $\boldsymbol{i}^{-1}$ **mod (p-1)**: $\boldsymbol{i}^{-1}$ **mod (p-1) exists** if $\mathbf{gcd}(\boldsymbol{i}, \boldsymbol{p} - \mathbf{1}) = \mathbf{1}$, i.e. ***i*** and **p-1** are relatively prime.

$\mathbf{k^{-1}}$ can be found using either Extended Euclidean algorithmt or Euler theorem or …..

>> **i_m1=mulinv(i,p-1)** % i^{-1}mod (**p**-1) computation.

- Compute **r=g$^{\mathbf{i}}$ mod p**
- Compute **s=(h-xr)i^{-1} mod (p-1) --> h=xr+is mod (p-1)**

Signature **σ=(r, s)**

$$\left.\begin{aligned} s &= (h - x\cdot r)\cdot i^{-1} \quad / i \\ s\cdot i &= (h - x\cdot r)\cdot \underline{\cancel{i^{-1}}\cdot\cancel{i}} \\ h - xr &= s\cdot i \longrightarrow h = x\cdot r + i\cdot s \end{aligned}\right\} \bmod p$$

2.***Signature Verification***

A signature **σ=(*r, s*)** on a h-value ***h*** of message ***M*** is verified using Public Parameters **PP**=(**p**, **g**) and **PuK$_{\mathbf{A}}$=a**.

1. Bob computes **h=H(M)**.
2. Bob verifies if **1<r<p-1** and **1<s<p-1.**
3. Bob calculates **V1=g$^{\mathbf{h}}$ mod p** and **V2=a$^{\mathbf{r}}$r$^{\mathbf{s}}$ mod p,** and verifies if **V1=V2.**

The verifier Bob accepts a signature if all **conditions** are satisfied during the signature creation and rejects it otherwise.

3.**Correctness**

The algorithm is correct in the sense that a **signature generated with the signing algorithm will always be accepted by the verifier**.
The signature generation implies

h=xr+is mod (p-1)

Hence Fermat's little theorem implies that all operations in the exponent are computed mod (**p**-1)

$$\underbrace{g^{h} \bmod p}_{V1} = g^{(xr+is) \bmod (p-1)} \bmod p = g^{xr}g^{is} = (\underbrace{g^{x}}_{(a)})^{r}(\underbrace{g^{i}}_{(r)})^{s} = \underbrace{a^{r}r^{s} \bmod p}_{V2}$$

```
>> p= int64(268435019)
p = 268435019
```

```
>> i =int64(randi(p-1))
i = 201156232
```

```
>> r=mod_exp(g,i,p)
r = 172536234
```

```
>> g_h=mod_exp(g,h,p)
g_h = 241198023
```

```
>> p= int64(268435019)
p = 268435019
>> g=2;
>> x =int64(randi(p-1))
x = 65770603
>> a=mod_exp(g,x,p)
a = 232311991
>> M='Hello Bob...'
M = Hello Bob...
>> h=hd28(M)
h = 150954921
```

```
>> i =int64(randi(p-1))
i = 201156232
>> gcd(i,p-1)
ans = 2
>> i =int64(randi(p-1))
i = 35395315
>> gcd(i,p-1)
ans = 1
>> i_m1=mulinv(i,p-1)
i_m1 = 192754179
>> mod(i*i_m1,p-1)
ans = 1
```

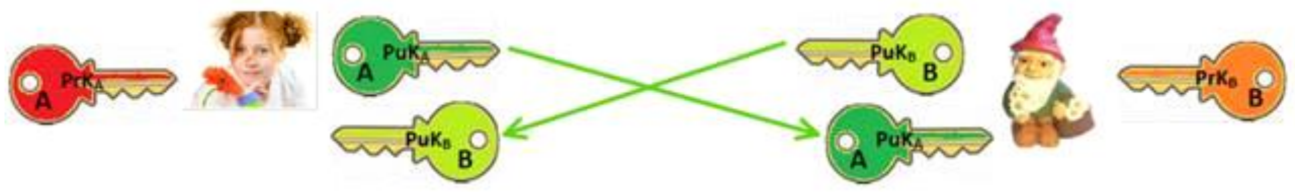
```
>> r=mod_exp(g,i,p)
r = 172536234
>> hmxr=mod(h-x*r,p-1)
hmxr = 20262153
>> s=mod(hmxr*i_m1,p-1)
s = 44575091
```

```
>> g_h=mod_exp(g,h,p)
g_h = 241198023
>> V1=g_h
V1 = 241198023

>> a_r=mod_exp(a,r,p)
a_r = 49998673
>> r_s=mod_exp(r,s,p)
r_s = 111993804
>> V2=mod(a_r*r_s,p)
V2 = 241198023
```

3. Identification.

If person can prove that he/she knows **PrK** corresponding to his/her **PuK** without revealing any information about **PrK** then everybody can trust that he is communicating with person posessing (**PrK**, **Puk**) key pair. This kind of proof is named as ***Zero Knowledge Proof*** (**ZKP**) and plays a very important role in cryptography. It is very useful to realize Identification, Digital Signatures and many other cryptographically secure protocols in internet. In many cryptographic protocols, especially in identification protocols **PrK** is named as **witness** and **PuK** as a **statement** for **PrK**. Every actor is having the corresponding key pair (**PrK$_A$**, **PuK$_A$**) and all **PuK** are exchanged between the users using open communication channel as indicated in figure below.
Let Bob is sure that **PuK$_A$** is indeed of Alice and wants to tell Alice that he intends to send her his photo with chamomile flowers dedicated for Alice. But he wants to be sure that he is communicating only with Alice itself and with nobody else. He hopes that at first Alice will prove him that she knows her secret **PrK$_A$** using **ZKP** protocol. In general, this protocol is named as **Identification protocol**, it is interactive and has 3 communications to exchange the following data named as ***commitment***, ***challenge*** and ***response***.



Registration phase: Bank generates **PrK$_A$** = **x** and **PuK$_A$** = **a** to Alice and hands over this data in smart card, or in other crypto chip in Alice's smart phone, or in software for Smart ID.

Schnorr Id Scenario: **Alice** wants to prove **Bank** that she knows her Private Key - **PrK$_A$** = **x** which corresponds to her Public Key - **PuK$_A$**= ***a*** not revealing **PrK$_A$**: Zero Knowledge Proof - **ZKP**
Protocol execution between **Alice** and **Bank** has time limit.
Alice's computation resources has a limit --> protocol must be computationally effective.
PrK$_A$=x is called a **witness** and corresponding **PuK$_A$**=a=g^x **mod** p is called a **statement**.
This protocol is initiated by Alice and has the following three communications.

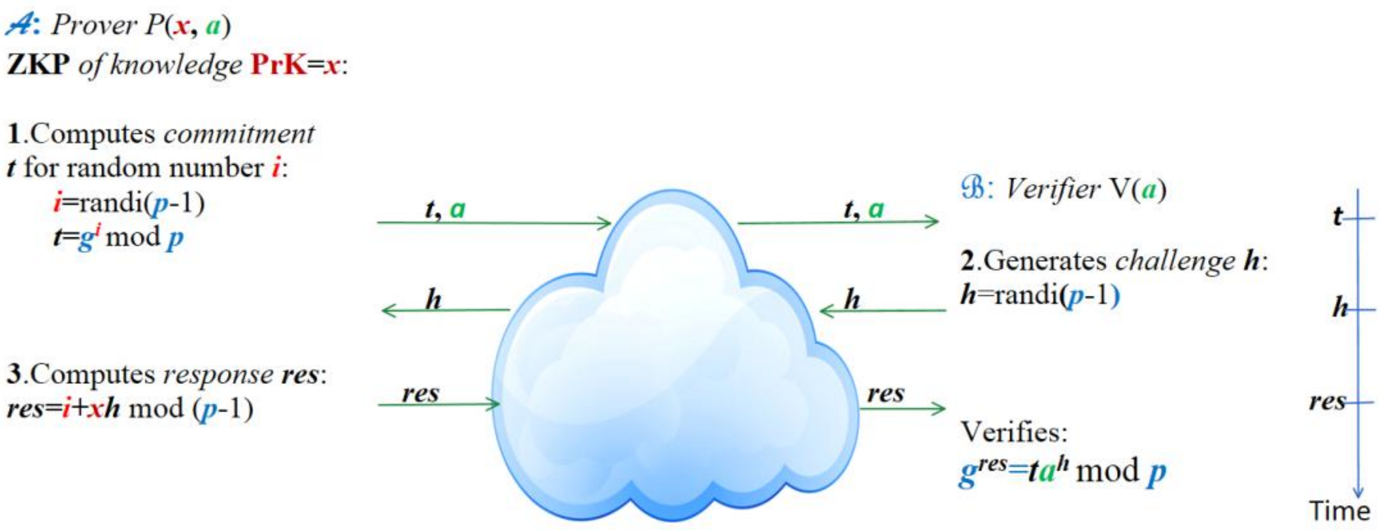
P(*x*, a) - Prover - Alice **V(a) - Verifier - Bank**

P(***x***, a) - **Prover** - **Alice** **V**(a) - **Verifier** - **Bank**

C:\Users\Eligijus\Documents\REKLAMA ZKP_Pasaka

4. Schnorr Identification: Zero Knowledge Proof - ZKP PP = (*p*, *g*).

Schnorr Id is interactive protocol, but not recurent as it is realized to prove the knowledge of mirackle words.
Schnorr Id Scenario: **Alice** wants to prove **Bank** that she knows her Private Key - $\mathbf{PrK_A} = x$ which corresponds to her Public Key - $\mathbf{PuK_A} = a = g^x$ **mod** p not revealing $\mathbf{PrK_A} = x$.



Correctness:

$$g^{res} \bmod p = g^{i+xh \bmod (p-1)} \bmod p = g^i g^{xh} \bmod p = t(g^x)^h \bmod p = ta^h \bmod p.$$

5. Schnorr Signature Scheme (S-Sig).

In general, to create a signature on the message of any finite length ***M*** parties are using cryptographic secure H-function (message digest).
In Octave we use H-function
>> hd28('…') % the input '…' of this function represents a string of symbols between the commas.
% the output of this function is decimal number having at most 28 bits.

Let ***M*** be a message in string format to be signed by **Alice** and sent to **Bob**: >> M='Hello Bob'
For signature creation **Alice** uses public parameters **PP=(*p*, *g*)** and
Alice's key pair is $\mathbf{PrK_A}=x$, $\mathbf{PuK_A}= a = g^x$ **mod** p.

Alice chooses at random u, $1<u<p-1$ and computes first component r of his signature:

$$r=g^u \bmod p. \quad (2.19)$$

Alice computes H-function value h and second component s of her signature:

$$h=\mathrm{H}(M||r), \quad (2.20)$$

$$s=u+xh \bmod (p-1). \quad (2.21)$$

Alice's signature on h is $\sigma=(r, s)$. Then **Alice** sends M and σ to **Bob**.

After receiving $\boldsymbol{M'}$ and $\boldsymbol{\sigma}$, **Bob** according to (2.20) computes $\boldsymbol{h'}$

$$\boldsymbol{h'}=\mathbf{H}(\boldsymbol{M'}\|\boldsymbol{r}),$$

and verifies if

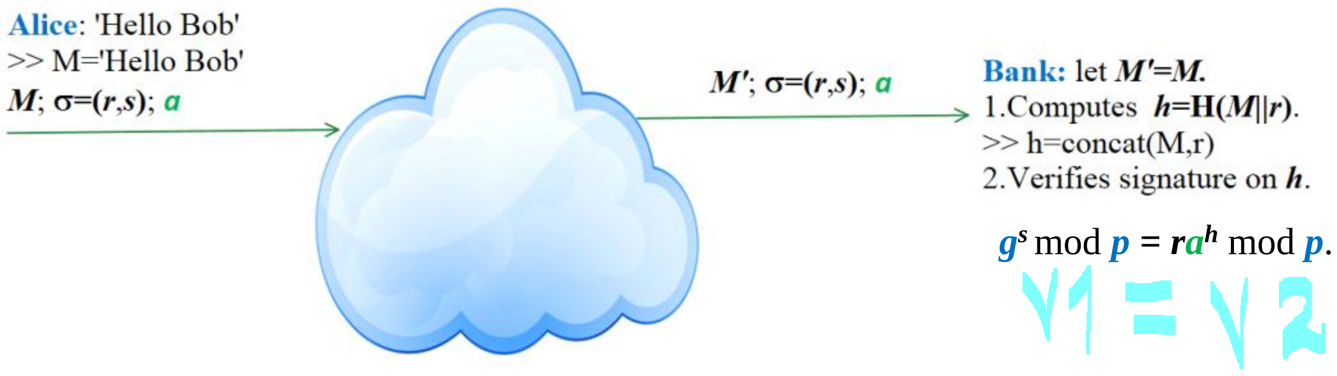
$$\underbrace{g^s \bmod p}_{V1} = \underbrace{ra^{h'} \bmod p}_{V2}. \qquad (2.22)$$

Symbolically this verification function we denote by

$$\mathbf{Ver}(a,\sigma,h')=\mathrm{V}\in\{True, False\}\equiv\{1, 0\}. \qquad (2.23)$$

This function yields ***True*** if (2.22) is valid if: $\boldsymbol{h}=\boldsymbol{h'}$ and $\mathbf{PuK_A}= a = F(\mathbf{PrK_A})= g^x \bmod p$.

and: $\boldsymbol{M}=\boldsymbol{M'}$



Correctness:

$$g^s \bmod p = g^{u+xh \bmod (p-1)} \bmod p = g^u g^{xh} \bmod p = r(g^x)^h \bmod p = ra^h \bmod p.$$

Non-Interactive ZKP — NIZKP

A: $u \leftarrow \mathrm{randi}(\mathbb{Z}_{p-1})$

$r = g^u \bmod p$

$h = H(a \| r)$ // Ethereum Keccak-256

$s = u + xh \bmod (p-1)$

NIZKP $= (r, s)$

$$\xrightarrow{(r,s),\ a}$$

B: $PuK_A = a$ $\quad h = H(a\|r)$

Verifies if

$$\boxed{g^s = ra^h \bmod p \quad V1 = V2}$$

correctness: $\overset{V1}{g^s \bmod p} = g^{u+xh \bmod (p-1)} \bmod p =$

$= g^u g^{xh} \bmod p = r(g^x)^h \bmod p = \underset{V2}{ra^h} \bmod p$

```
>> p= genstrongprime(28)
268435019
```

Example of signature realization with Octave

p - is strong prime if
p = 2*q + 1, when q - is prime.
Then q = (p-1)/2

```
>> p= int64(268435019);   % p is strong prime
>> g_s=mod_exp(g,s,p)
>> p= int64(268435019);   % p is strong prime
>> g=2;

>> x=int64(randi(p-1))
x = 89089011
>> a=mod_exp(g,x,p)
a = 221828624

>> m='Hello Bob'
m = Hello Bob
>> u=int64(randi(p-1))
u = 228451192
>> r=mod_exp(g,u,p)
r = 33418907
>> cc=concat(m,r)
cc = Hello Bob33418907      % cc is a string
% type variable
>> cc=concat(m,'33418907')
cc = Hello Bob33418907
>> ccc=concat(m,'r')
ccc = Hello Bobr

>> h=hd28(cc)
h = 104824510
>>  s=mod((u+x*h),p-1)
s = 147250342
```

104824510

$\sigma = (r, s)$

m, a

```
>> xh=mod(x*h,p-1)
.....
>>  s=mod((u+xh),p-1)
```

```
>> g_s=mod_exp(g,s,p)
g_s = 185672370
V1=g_s;
>> a_h=mod_exp(a,h,p)
a_h = 263774143
>>  V2=mod(r*a_h,p)
V2 = 185672370
```

p - is strong prime if
p = 2*q + 1, when q - is prime.
Then q = (p-1)/2

```
>> p= int64(268435019)
p = 268435019
>> isprime(p)
ans = 1
>> q=(p-1)/2
q = 134217509
>> isprime(q)
ans = 1
>>
>> pb=dec2bin(p)
pb =
1111111111111111111001001011
```