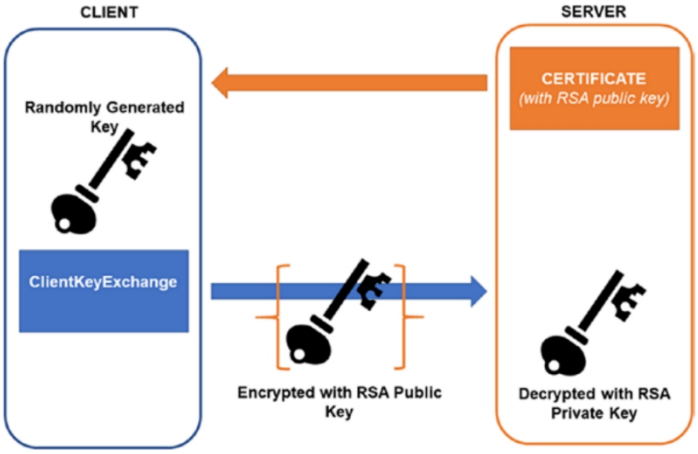
Laborator 5
Gestiunea cheilor și semnarea mesajelor bazată pe curbe eliptice

Utilizarea Diffie-Hellman(DH)]n gestiunea cheilor

După cum am văzut la curs abordarea DH este un pic diferită față de cea RSA. Acesta din urmă este folosit pentru criptarea și decriptarea mesajelor în timp ce DH se ocupa inițial numai de schimbarea cheilor. După cum știți deja RSA este dominant utilizat pentru transmiterea cheilor în medii nesigure. În figura de mai jos este prezentată o utilizare a acestuia în TLS (despre care am discutat deja) pentru stabilirea conexiunii.

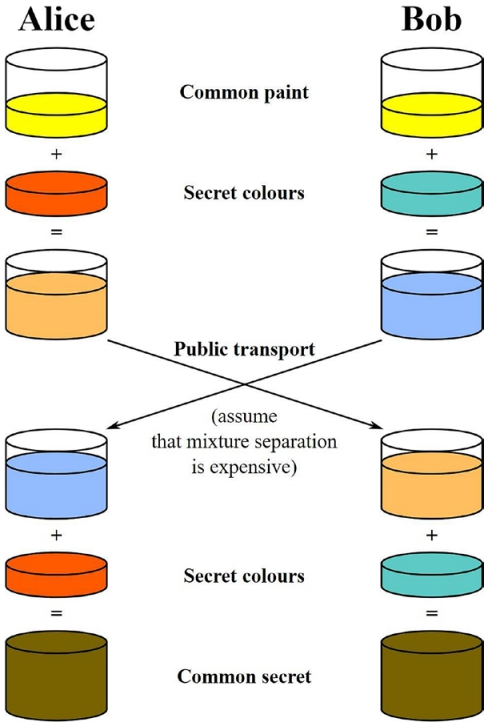


Se observă că clientul poate genera o cheie de sesiune aleatoare pe care o criptează cu cheia publică a serverului și apoi o utilizează. Acest proces garantează și faptul că serverul se află în posesia unui certificat valid deoarece numai acesta poate ulterior decripta cheia sesiune pentru a o utiliza în comunicare.

NU uitați ca TLS nu autentifică de obicei clienții. Totuși dacă se cere autentificarea clientului acesta trebuie sa dovedească (independent) faptul că este posesorului unui certificat prin semnarea unui “nounce” de la server.

Pe de altă parte DH și versiunea acestuia bazată pe curbe eliptice ECDH pot crea instantaneu chei fără a schimba vreo informație secretă între părțile implicate în proces. Pur și simplu se schimbă între ei parametri publici care le permit ca simultan părțile implicate să calculeze aceeași cheie. Acest proces se numește agrearea cheii.

La început DH crează o pereche de numere unul privat și unul public pentru fiecare participant la discuție. Apoi partenerii pun la dispoziție cheile publice. Cheia de la vecin împreună cu cheia locală privată vor fi utilizate pentru crearea unui secret cunoscut de toți participanții. Pentru înțelegerea procesului mai jos aveți o reprezentare intuitivă.



Trebuie totuși făcută observația că spre deosebire de RSA DH și ECDH nu permit transmiterea oricărui tip de date. Ca rezultat acestea pot conveni asupra unei date aleatoare și nu aleg un mesaj anume. Acesta poate și este de obicei folosit ca o cheie master simetrică din care se pot deriva ulterior alte chei de același tip. Mai mult schimbul trebuie să fie reciproc nu unidirecțional. Deci sunt situații (rare) în care deoarece nu pot avea comunicare bidirecțională nu ne rămâne decât să utilizam RSA. Mai jos aveți un program care pune în evidență schimbul de chei specific DH. Nu uitați să introduceți în proiect modulul cryptography.

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import dh
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.backends import default_backend

if __name__ == '__main__':
    # se genereaza niste parametri reutilizabili
    # ar trebuie sa fie aceasi si pentru celalalt capat
    parameters = dh.generate_parameters(generator=2, key_size=1024,
                                        backend=default_backend())
    # se genereaza chei privata utilizata in timpul schimbului
    private_key = parameters.generate_private_key()
    # de remarcat ca intr-o situatia reala de stabilirea acomunicatiei cheia peer_public_key
    # va fi receptionata din alta parte/locatie. De exemplu se va genera alta cheie privata si apoi vom
    #cere/primi
    # cheia publica de la celalat capat. Nu uitati ca partile trebuie totusi initial sa cada de acord asupra unui
    # set comun de parametri
    peer_public_key = parameters.generate_private_key().public_key()
    shared_key = private_key.exchange(peer_public_key)
    # se deriva cheia
    derived_key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=b'handshake data',
        backend=default_backend()
    ).derive(shared_key)

    # test primar
    if len(derived_key) == 32:
        print("[Merge]")
    else:
        print("[Nu merge!]")
```

Tema1. Măsurați câte ms durează tot programul apoi creați o aplicație client server simplă care realizează schimbul de chei DH apoi utilizează secretul obținut pentru a cripta simetric canalele de emisie recepție (cu ce vreți voi din biblioteca de criptare (e.g. AES))

DH nu este ceva deosebit, are și el problemele lui. În primul rând sunt numai două valori legale pentru generator și anume 2 și 5 ceea ce este cel puțin ciudat pentru un protocol criptografic unde alegerea generatorului nu contează din motive de securitate ci trebuie să fie doar identică la ambele capete. De aceea trebuie să învățați și să înțelegeți teoria și să citiți standardele recomandate de care fugiți ca dracul de tămâie.
Dimensiunea cheii este importantă și ar trebui pentru chestiuni curente să fie de minim 2048 biți (vezi standardele). Deși în general toată lumea spune ca DH are o generare de chei rapidă acest lucru nu prea este

corect. Din contră generarea parametrilor utilizați în generarea cheii poate fi lentă și ca atare în utilizare per ansamblu poate fi destul de puturos.
Această problemă poate fi diminuată global dacă acești parametri sunt reutilizați pentru generarea mai multor chei (ceea ce nu este sănătos dacă avem constrângeri mai dure de securitate) dar pentru magazine electronice pentru vândut floricele. Dacă însă treaba este mai serioasă deoarece nu mai avem ce face și atunci trebuie să utilizăm variația lui DH bazată pe curbe eliptice care este mult mai rapidă.

DH bazat pe curbe eliptice - Elliptic-Curve Diffie-Hellman (ECDH)
Despre curbe eliptice am discutat deja la curs astfel încât datorită modificărilor minimale vom prezenta direct programul primar de test

```
 from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives.kdf.hkdf import HKDF

if __name__ == '__main__':
    # se genereaza chei privata utilizata in timpul schimbului
    private_key = ec.generate_private_key(
        ec.SECP384R1(), default_backend()
    )
    # de remarcat ca intr-o situatia reala de stabilirea acomunicatiei cheia peer_public_key
    # va fi receptionata din alta parte/locatie. Aici vom genera o alta cheie privata si
    # vom obtine cheia publica pentru ea
    peer_public_key = ec.generate_private_key(
        ec.SECP384R1(), default_backend()
    ).public_key()
    shared_key = private_key.exchange(ec.ECDH(), peer_public_key)
    # se deriva cheia
    derived_key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=b'handshake data',
        backend=default_backend()
    ).derive(shared_key)
    # test primar
    if len(derived_key) == 32:
        print("[Merge]")
    else:
        print("[Nu merge!]")
```

Crearea cheilor utilizând DH/ECDH este preferată în multe cazuri în locul abordării RSA. Există multe motive dar unul dintre cele mai importante este legat de trimiterea secretului (“forward secrecy”).

Tema 2. Reluați Tema1 dar de această dată utilizați ECDH. Comparați timpii de execuție între cele două soluții.

DH și trimiterea secretului
După cum am discutat DH/ECDH nu au autentificare. Aceasta se datorează faptului că cheile sunt complet aleatoare și temporare și astfel nu avem posibilitatea să le asociem unei identități electronice. Din acest motiv în multe cazuri abordările bazate pe DH/ECDH au nevoie de o cheie publică de durată (persistentă) bazată pe RSA sau ECDSA care este protejată de de un certificat. Aceste chei nu sunt utilizate în criptarea

datelor sau transportul cheii sigurul lor scop fiind să stabilească clar identitatea fiecărei părți implicate în comunicare și eventual (vezi TLS) să fie utilizate în gestiunea unei provocări sau a unui nounce.
Nu uitați faptul că parametrii DH trebuie regenerați pentru fiecare schimb de chei. Biblioteca cryptography dă în documentație câteva exemple dar care nu acoperă și astfel de aspecte.
Programul de mai jos salvează ceea ce ar trebui sa fie o cheie pentru o singura utilizare care să fie ulterior utilizată. Nu uitați să distrugeți cheile după utilizare (eroarea comună este să le scăpați în zona de jurnalizare - din motive de depanare).

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives.kdf.hkdf import HKDF

if __name__ == '__main__':
    class ECDHExchange:
        def __init__(self, curve):
            self._curve = curve
            # generati o cheie privata temporara care va fi utilizata in procesul de schimb de date
            self._private_key = ec.generate_private_key(
                curve, default_backend())
            self.enc_key = None
            self.mac_key = None
        def get_public_bytes(self):
            public_key = self._private_key.public_key()
            raw_bytes = public_key.public_bytes(
                encoding=serialization.Encoding.PEM,
                format=serialization.PublicFormat.SubjectPublicKeyInfo)
            return raw_bytes
        def generate_session_key(self, peer_bytes):
            peer_public_key = serialization.load_pem_public_key(
                peer_bytes,
                backend=default_backend())
            shared_key = self._private_key.exchange(
                ec.ECDH(),
                peer_public_key)
            # generatie sursa primara pe 64 biti pentru a putea crea doua chei de 32 de biti
            key_material = HKDF(
                algorithm=hashes.SHA256(),
                length=64,
                salt=None,
                info=None,
                backend=default_backend()).derive(shared_key)
            # obtinerea cheii de criptare
            self.enc_key = key_material[:32]
            # generare cheie MAC
            self.mac_key = key_material[32:64]
    # verificare buna functionare
    client_x = ECDHExchange(ec.SECP384R1())
    client_y = ECDHExchange(ec.SECP384R1())
    client_x.generate_session_key(client_y.get_public_bytes())
    client_y.generate_session_key(client_x.get_public_bytes())
    if client_x.enc_key and client_x.mac_key and client_x.enc_key == client_y.enc_key and
client_x.mac_key == client_y.mac_key:
        print("In regula")
    else:
        print("[Eroare]")
```

Se observă că pentru a efectua schimbul ECDH ambele entități implicate în comunicare vor instanția clasa și apoi vor apela metoda get_public_bytes pentru a obține datele care trebuie trimise celeilalte părți. La recepția acestor date ele sunt utilizate pentru generarea unei chei sesiune_session_key din care apoi prin deserializare se obține o cheie publică care este folosită pentru crearea cheii comune.
Dacă ne uităm în program se observă ceva numit HKDF. Aici se generează subchei pornind de la una primară/inițială iar abordarea este foarte utilizată în comunicațiile din rețea.
Atenție ea nu trebuie utilizată pentru păstrarea/securizarea datelor. În exemplu aceasta este utilizată pentru a obține (pornind de la cheia comună) o cheie de criptare precum și o cheie pentru autentificarea mesajului - Message Authentication Code(MAC) prin împărțirea cheii primare în doua subchei de dimensiune egală. Într-un caz real această funcție este mai complicată (după cum ați învățat în facultate) dar pentru simplitatea exemplului s-a ales această abordare.
În acest exemplu se observă că realizăm schimbul de chei temporare dar în lipsa autentificării nici o parte nu poate fi sigură de identitatea celeilalte. Pentru a rezolva problema vom modifica acest exemplu astfel încât să avem și autentificarea părților implicate utilizând și o cheie asimetrică pentru semnarea datelor.

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
import struct    # este necesara pentru implementarea lui get_signed_public_pytes

class AuthenticatedECDHExchange:
    def __init__(self, curve, auth_private_key):
        self._curve = curve
        self._private_key = ec.generate_private_key(
            self._curve,
            default_backend())
        self.enc_key = None
        self.mac_key = None
        self._auth_private_key = auth_private_key
    # Part of AuthenticatedECDHExchange class
    def get_signed_public_bytes(self):
        public_key = self._private_key.public_key()
        # acici aven sirul brut (raw) de octeti
        raw_bytes = public_key.public_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PublicFormat.SubjectPublicKeyInfo)
        # aici avem semnatura care ne identifica pe noi (emitator de exemplu)
        signature = self._auth_private_key.sign(
            raw_bytes,
            ec.ECDSA(hashes.SHA256()))

        # deoarece dimensiunea semnaturii este variabila a fost necesara utilizarea lui len
        return struct.pack("I", len(signature)) + raw_bytes + signature
    def generate_session_key(self, peer_bytes, signature_pub_key):
        peer_key_signature_len, = struct.unpack("I", peer_bytes[:4])
        public_key_bytes = peer_bytes[4:-peer_key_signature_len]
        public_key_signature = peer_bytes[-peer_key_signature_len:]
        signature_pub_key.verify(
            public_key_signature,
            public_key_bytes,
            ec.ECDSA(hashes.SHA256()))
        peer_public_key = serialization.load_pem_public_key(
            public_key_bytes,
            backend=default_backend())
        shared_key = self._private_key.exchange(
```

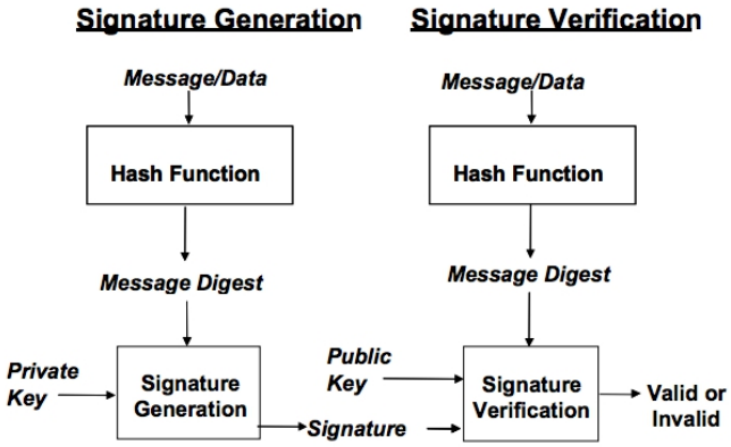
```
                ec.ECDH(),
                peer_public_key)
            # obtinem cheia primara
            key_material = HKDF(
                algorithm=hashes.SHA256(),
                length=64,
                salt=None,
                info=None,
                backend=default_backend()).derive(shared_key)
            # generam cheia de criptare
            self.enc_key = key_material[:32]
            # generam cheia MAC
            self.mac_key = key_material[32:64]
if __name__ == '__main__':
    # zona de testare a clasei
    private_key_x = ec.generate_private_key(ec.SECP384R1(), default_backend())
    private_key_y = ec.generate_private_key(ec.SECP384R1(), default_backend())
    client_x = AuthenticatedECDHExchange(ec.SECP384R1(), private_key_x)
    client_y = AuthenticatedECDHExchange(ec.SECP384R1(), private_key_y)
    client_x.generate_session_key(client_y.get_signed_public_bytes(), private_key_y.public_key())
    client_y.generate_session_key(client_x.get_signed_public_bytes(), private_key_x.public_key())
    if client_x.enc_key and client_x.mac_key and client_x.enc_key == client_y.enc_key and
client_x.mac_key == client_y.mac_key:
        print("In regula")
    else:
        print("[Eroare!]")
```

Dacă ne uităm la program observăm că clasa ECDHExchange a fost redenumită ca AuthenticatedECDHExchange, iar constructorul a fost modificat pentru a putea prelua o cheie privată persistentă care va fi utilizată pentru semnare. Se observă o diferență între _private_key, care este generată și temporară și auth_private_key care este persistentă și trimisă ca un parametru pentru stabilirea identității. Aici se putea utiliza o cheie RSA dar pentru că discutăm de utilizarea curbelor eliptice în DH am utilizat o astfel de curbă (adică o cheie ECDSA). Se observă că putem genera șiruri de octeți care să fie semnate public. Mai departe la destinație întâi trebuie despachetați primii patru octeți pentru a obține lungimea semnăturii. Aceasta va fi verificată utilizând cheia proprie publică persistentă (cam ca la RSA). Dacă se trece de aceasta verificare atunci putem considera în limite rezonabile că am primit parametrii ECDH de la cine trebuie.

Tema 3 Testați programul și înlocuiți cheia ECDSA cu una RSA apoi evident retestați programul. Faceți și măsurători de viteză.

Cum semnăm ECDSA o tranzacție?

Despre aspectele teoretice am discutat la curs. Mai jos o să prezentăm o metodă specifică bibliotecilor de Ethereum/Bitcoin. În figură avem fluxul de semnare și de verificare conform acestei abordări



După cum se observă procesul se poate automatiza destul de ușor deci poate fi efectuat de portofelul virtual de criptomonede în locul utilizatorului. În acest exemplu emițătorul va semna un mesaj oarecare cu o cheie, va împacheta mesajul cu ajutorul semnăturii. La destinație se va face verificarea mesajului cu ajutorul cheii de verificare.
Pentru a semna un mesaj (sau în mod particular o tranzacție) se pot utiliza parametri pe 192 de biți (vezi curs) și putem mai concret să ne bazăm de exemplu pe parametrii specificați în secp192k1. Aceștia fac parte dintr-o colecție de standarde industriale dezvoltate de către grupul de standarde pentru criptografie eficientă — Standards for Efficient Cryptography Group (SECG) - www.secg.org. Acesta din urmă pare că și-a încetat activitatea dar materialele sunt clare și din nefericire încă de actualitate.
Pentru a dezvolta programul de test sunt necesare o serie de aplicații și biblioteci pe care, în caz că nu le aveți, le puteți instala cu:

```
apt install gcc python3-dev libgmp3-dev
```

Apoi vom crea un proiect in pycharm. Aici, mai intai, adaugati modulul pycryptodome la proiect de unde din hash a lui cypto vom utiliza keccack, ne mai trebuie si pachetul fastecdsa. Acum putem importa pe keccack

```
from Crypto.Hash import keccak
```

Apoi se pot adăuga parametrii curbei eliptice extrași din standardul secp192k1 numiți “Pcurve” și cofactorul N pe care îi vom folosi. Ambii parametri sunt de fapt numere prime de mari dimensiuni care vor fi folosite în cheia privata care va fi intre 1 si N.

```
Pcurve = 2**256-2**32-2**9-2**8-2**7-2**6-2**4-1
N = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141
Ac = 0
Bc = 7
Gx = 0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798
Gy = 0x483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8
```

Împreună cu anteriorii aceste variabile vor fi și ele folosite în generarea cheilor publice și private. După cum știți deja din facultate pentru operațiile criptografice sunt necesare și funcția de cel mai mare divizor comun din algoritmul euclidean extins împreună cu inversul modulului (atenție tot cel specific criptografiei). Pentru ajutor aveți mai jos aceste funcții.

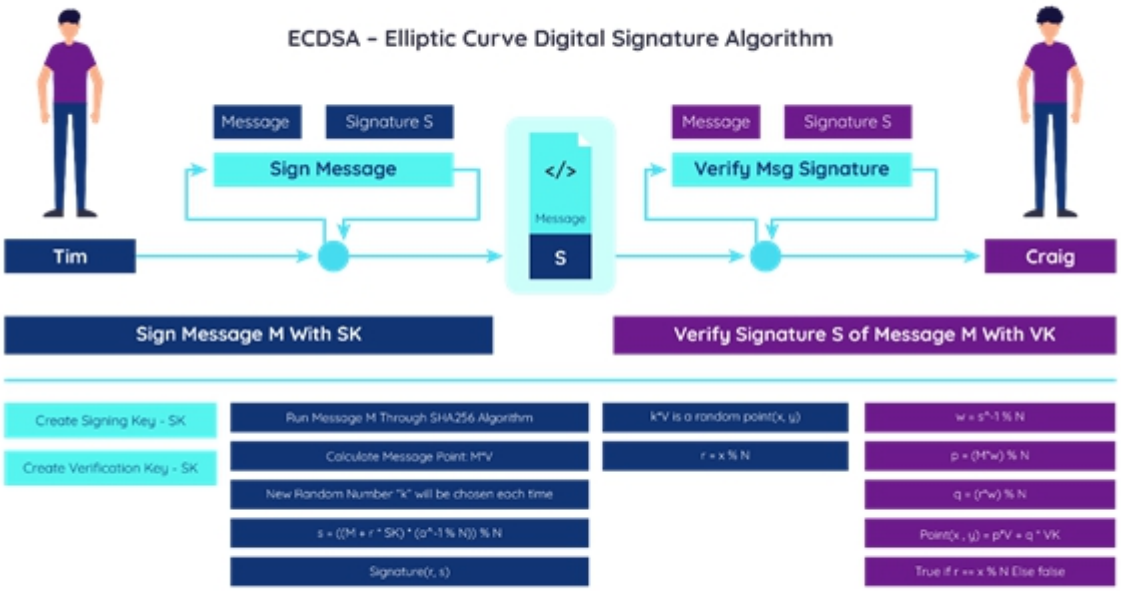
```
def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, y, x = egcd(b % a, a)
        return (g, x - (b // a) * y, y)

def modinv(a, m):
    g, x, y = egcd(a, m)
```

```
    if g != 1:
        raise Exemption('inversul modulului nu exista')
    else:
        return x % m
```

Ne mai trebuie adăugate o serie de module la proiect și ca atare

```
from fastecdsa.curve import Curve
from fastecdsa.point import Point
from fastecdsa.curve import secp256k1
```

Procesul de semnare și de verificare este prezentat în detaliu mai jos.



Acum putem semna un mesaj de exemplu 'Nu tot ce este blochain zboara!'

```
keccak.new(data=b'Nu tot ce este blochain zboara!', digest_bits=256).hexdigest()
```

Acum am obținut funcția hash asociată mesajului. Mai departe vom crea o variabilă numita msgHash care să rețină reprezentarea hexa.

```
msgHash = '0x' + keccak.new(data=b'I LOVE BLOCKCHAIN', digest_bits=256).hexdigest()
```

Ne mai trebuie creat și un punct generator care va fi utilizat la verificarea cheii. Practic cheia de verificare se va obține prin înmulțirea cheii de semnare cu punctul de verificare.

```
GenPoint = Point(Gx, Gy, curve=secp256k1)
```

Dacă vom afișa acest punct vom observa că este pe curba utilizată de Bitcoin și Ethereum. Acum trecem la crearea cheilor pentru semnare și verificare. Cheia pentru semnare este generată prin alegerea unui număr aleator (nu uitați despre importanța conceptului) care trebuie să fie mai mic ca N. Apoi vom crea cheia de verificare prin înmulțirea cheii de semnare cu punctul de pe curbă.

```
GenPoint = Point(Gx, Gy, curve=secp256k1)
SK = 99900007777333333333333333333333
VK = SK *GenPoint
```

Acum avem nevoie de X și Y pentru un al doilea punct care trebuie să fie un număr aleator. Nu uitați să importați modulul random.

```
random.randint(99999, 999999999999)
randNum = random.randint(99999, 999999999999)
```

În sfârșit putem crea punctul XY1 și pe r:

```
XY1 = randNum*GenPoint
r=XY1.x % N
```

Trebuie să îl creăm și pe s numai că de data asta vom utiliza mesajul ca pe un întreg.

```
msgHash = int(msgHash, base=16) # convertesc sirul in int
s = ((msgHash + r * SK) * modinv(randNum, N)) % N
```

Acum avem semnătura(r,s)
În sfârșit putem începe procesul de verificare ECDSA. Condiția de verificare este r = x % N iar de fiecare dată când dorim să creăm o nouă semnătură va trebui sa creăm un nou k dintr-un punct aleator ales k*V de pe curba eliptica selectată.
Pentru a putea începe verificările ne mai trebuie o serie de parametri după cum urmează (nu uitați luați și citiți standard-ul pentru detalii).

```
w = modinv(s, N)
u1 = msgHash * w % N
u2 = r * w % N
XY2 = u1 * GenPoint + u2 * VK
```

Acum se poate realiza o verificare dacă condiția `r == XY2.x % N` este adevărată.

Și codul pentru testare

```
from Crypto.Hash import keccak
from fastecdsa.curve import Curve
from fastecdsa.point import Point
from fastecdsa.curve import secp256k1
import random

def println(msg,i):
    print(msg+i+'\n')
def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, y, x = egcd(b % a, a)
        return (g, x - (b // a) * y, y)
def modinv(a, m):
    g, x, y = egcd(a, m)
    if g != 1:
        raise Exemption('nu exista inversul modulului')
    else:
        return x % m
if __name__ == '__main__':
    Pcurve = 2 ** 256-2 ** 32-2 ** 9-2 ** 8-2 ** 7-2 ** 6-2 ** 4-1
    N = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141
    Ac = 0
    Bc = 7
```

```
    Gx = 0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798
    Gy = 0x483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8
    keccak.new(data=b'Nu tot ce este blochain zboara!', digest_bits=256).hexdigest()
    msgHash = '0x' + keccak.new(data=b'Nu tot ce este blochain zboara!', digest_bits = 256).hexdigest()
    println('Hash-ul mesaj:',msgHash)
    GenPoint = Point(Gx, Gy, curve=secp256k1)
    println('Punctul generat X    (de pe curba)',hex(GenPoint.x))
    println('Punctul generat Y    (de pe curba)', hex(GenPoint.y))
    println('N=',str(N))
    SK = 99900007777333333333333333333333
    println('SK=', str(SK))
    VK = SK *GenPoint
    println('VK=', str(VK))
    random.randint(99999, 999999999999)
    randNum = random.randint(99999, 999999999999)
    XY1 = randNum*GenPoint
    r=XY1.x % N
    println('Semnatura r=', str(r)) #semnatura(s,r)
    msgHash = int(msgHash, base=16) # convertesc sirul in int
    println('Hash-ul mesajului=', str(msgHash))
    s = ((msgHash + r * SK) * modinv(randNum, N)) % N
    println('Semnatura s=', str(s))
    w = modinv(s, N)
    println('w=', str(w))
    u1 = msgHash * w % N
    println('u1=', str(u1))
    u2 = r * w % N
    println('u2=', str(u2))
    XY2 = u1 * GenPoint + u2 * VK
    println('XY2=', str(XY2))
    if (r == XY2.x % N):
        println('Verificarea finala a semnaturii - ok','')
    else:
        println('Verificarea finala a semnaturii - eroare', '')
```

Tema 4 Testați programul - măsurați viteza de execuție (desigur fără print-uri).

Tema 5 Creați un arbore Merkle cu hash-urile unor fișiere dintr-un subdirector, verificați din timp în timp și anunțați pe baza recalculării hash-ului dacă s-au modificat fișierele.

Temă pe acasă
Rescrieți utilizând golang, rust, cpp sau chiar java/.net o parte din teme și faceți analize comparative cu privire la profilul execuției aplicației (memorie, viteză etc).

Temă pentru insomniaci
Încercați să creați un portofel rudimentar pentru criptomonede bazat pe biblioteca ethereum