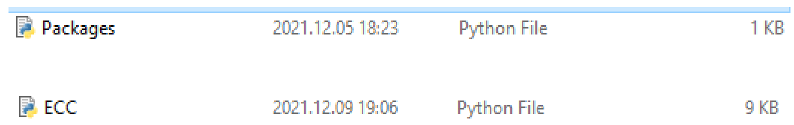
V, r, s

Key generation
1.Install Python 3.9.1.
2.Launch script Packages for joining a libraries.
3.Launch file ECC.
4.If window is escaping, then open hiden windows
in icon near the Start icon.



Elliptic Curve Digital Signature Algorithm - ECDSA

ECDA Public Parameters: **PP** = (***EC***, ***G***, ***p***), ***G***=(x_G, y_G); ElGamal CS Public Parameters: **PP** = (***p***, ***g***)
$1<x_G<$**n**, $1<y_G<$**n**.
n - is an order (number of points) of EC, i.e. according to **secp256k1** standard is equal to ***p***: **n**=***p***;
|**n**|=|***p***|=256 bits.
PrK_A=**z** <-- randi; **z**< **n**, max|**z**|<=256 bits.
PuK_A=**z******G***=***A***=(x_A, y_A); max|***A***|=2•256=512 bits.

Signature creation for message *M*

Signature is formed on the h-value ***h*** of Hash function of ***M***.
Recommended to use SHA256 algorithm

1. ***h*** = H(***M***)=SHA256(***M***);
2. ***t*** <-- randi; |***t***|$\leq$ 256 bits;
3. $R = t*G = t*(x_G, y_G) = (x_R, y_R)$;
4. $r = x_R \bmod p$;
5. $s = (h + z \bullet r) \bullet t^{-1} \bmod p$; $|s| \leq 256$ bits; // Since ***p*** is prime, then exists $s^{-1} \bmod p$. **σ**
   // >> s_m1=mulinv(s,p) % in Octave
6. **Sign**(PrK_{ECC}=***z***, ***h***) = **σ** = (***r***, ***s***)

Signature vrification: Ver(PuK, σ, *h*)

1. Calculate $u_1 = h \bullet s^{-1} \bmod p$ and $u_2 = r \bullet s^{-1} \bmod p$
2. Calculate the curve point $V = u_1 * G + u_2 * A = (x_V, y_V)$
3. The signature is valid if $R=V$; $r=x_V=x_R \bmod p$.

ECDSA	Schnorr Signature
h = H(***m***);	***h*** = H(***m***);
t ←randi;	***t*** ←randi;

$R = t*G = t*(x_G, y_G) = (x_R, y_R)$; $r = x_R \bmod p$; $\lvert t\rvert \le 256$ bits;	$r=g^t \bmod p$;
$s=(h+z\bullet r)t^{-1} \bmod p$; $\lvert s\rvert \le 256$ bits;	$s=(h+x\bullet t) \bmod (p-1)$;
Sig($PrK_{ECC}=z$, h) = $(r, s) = \sigma$;	Sig($PrK=x$, h) = $(r, s) = \sigma$;
$s^{-1}=(h+z\bullet r)^{-1}t \bmod p$;	
ECDSA Verification	**Schnorr Signature Verification**
Compute $u_1=h\bullet s^{-1} \bmod p$ and $u_2=r\bullet s^{-1} \bmod p$;	Compute $u_1= ra^h \bmod p$ and $u_2= g^s \bmod p$.
Compute $R = u_1*G + u_2*A = (x_R, y_R)$;	Signature is valid if: $u_1= u_2$
The signature is valid if $r=x_R \bmod p$.	The signature is valid if $u_1= u_2$.

Correctness: Can be Omitted

$R=u_1*G + u_2*A$

From the definition of the Public Key $A=z*G$ we have:

$R=u_1*G + (u_2\bullet z)*G$

Because EC scalar multiplication distributes over addition we have:

$R=(u_1 + u_2\bullet z)*G$

Expanding the definition of u_1 and u_2 from verification steps we have:

$R=(h\bullet s^{-1} + r\bullet s^{-1}\bullet z)*G$

Collecting the common term s^{-1} we have:

$R=[(h + r\bullet z)\bullet s^{-1}]*G$

Expanding the definition of s from signature creation we have:

$R=[(h + r\bullet z)\bullet(h + r\bullet z)^{-1}\bullet t]*G=t*G$.

Since the inverse of an inverse is the original element, and the product of an element's inverse and the element is the identity, we are left with $R = t*G = (x_R, y_R)$; $r=x_R$.

Ethereum for signing transactions is using **secp256k1** EC together with **keccak256** H-function.
secp256k1 has co-factor=1. When the cofactor is 1, everything is fine.
The signature of transaction in **Ethereum** is placed in the varaibles **v**, **r**, **s**.
Variable **v** represents the version of signature and (**r**, **s**)=σ.

Public-key cryptography is based on the intractability of certain mathematical problems.
Early public-key systems are secure assuming that it is difficult to factor a large integer composed of two or more large prime factors.
For elliptic-curve-based protocols, it is assumed that finding the discrete logarithm of a random elliptic curve element with respect to a publicly known base point (generator) is infeasible: this is the "elliptic curve discrete logarithm problem" (ECDLP).
The security of elliptic curve cryptography depends on the ability to compute a point multiplication and the inability to compute the multiplicand given the original and product points.
The size of the elliptic curve determines the difficulty of the problem.

The primary benefit promised by elliptic curve cryptography is a smaller key size, reducing storage and transmission requirements, i.e. that an elliptic curve group could provide the same level of security afforded by an RSA-based system with a large modulus and correspondingly larger key: for example, a 256-bit elliptic curve public key should provide comparable security to a 3072-bit RSA public key.
The U.S. National Institute of Standards and Technology (NIST) has endorsed elliptic curve cryptography in its Suite B set of recommended algorithms, specifically elliptic curve Diffie–Hellman (ECDH) for key exchange and Elliptic Curve Digital Signature Algorithm (ECDSA) for digital signature.
The U.S. National Security Agency (NSA) allows their use for protecting information classified up to top secret with 384-bit keys.[2]
However, in August 2015, the NSA announced that it plans to replace Suite B with a new cipher suite due to concerns about quantum computing attacks on ECC.[3]

https://en.wikipedia.org/wiki/SHA-2

SHA-2 (**Secure Hash Algorithm 2**) is a set of cryptographic hash functions designed by the United States National Security Agency(NSA).[3] Cryptographic hash functions are mathematical operations run on digital data; by comparing the computed "hash" (the output from execution of the algorithm) to a known and expected hash value, a person can determine the data's integrity.

~~SHA-160~~

$2^{\sqrt{160}} = 2^{80}$

2^{70}

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

2^{128} 2^{256}

Key Agreement Protocol - KAP

DH - Classic

$PP = (p, g)$

A: $u \leftarrow$ randi

$K_A = g^u \bmod p$

$\xrightarrow{K_A}$

B:

$v \leftarrow$ randi

$K_B = g^v \bmod p$

$\xleftarrow{K_B}$

A:

$(K_B)^u \bmod p = (g^v)^u \bmod p$ $(K_A)^v \bmod p =$

DH - EC

$PP = (EC, G, p)$

A: $u \leftarrow$ randi

$K_A = (u * G)$

$\xrightarrow{K_A}$

B:

$v \leftarrow$ randi

$K_B = (v * G)$

$\xleftarrow{K_B}$

A: $K_{AB} = u * K_B =$

B: $K_{BA} = v * K_A =$

$(k_B)^u \bmod p = (g^v)^u \bmod p$

$g^{vu} \bmod p = K_{AB}$

$(k_A)^v \bmod p =$

$= (g^u)^v \bmod p =$

$= g^{uv} \bmod p =$

$= K_{BA}$

$K_{AB} = k = K_{BA}$

$A: K_{AB} = u * K_B =$

$= u*(v*G) =$

$= (u \cdot v)*G =$

$= uv * G.$

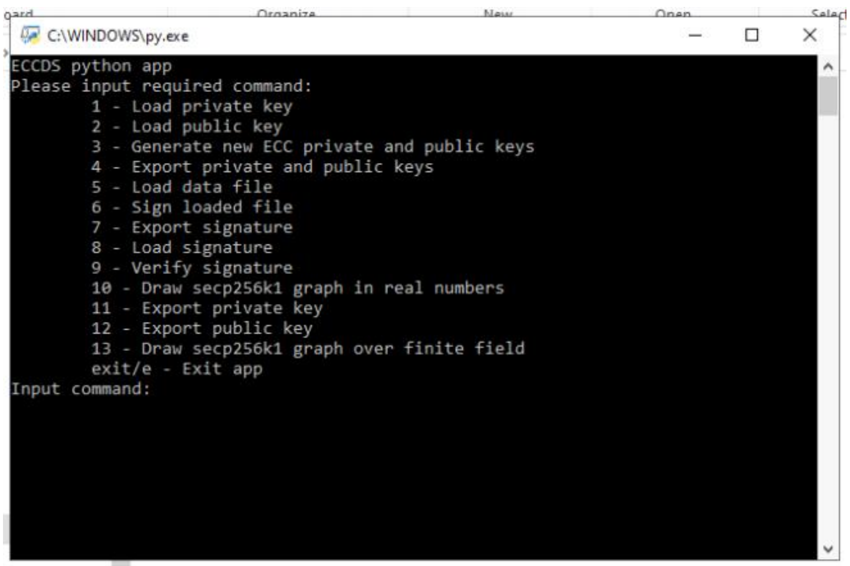
$B: K_{BA} = v * K_A =$

$= v*(u*G) =$

$= (v \cdot u)*G =$

$= vu * G$

$K_{AB} = K = K_{BA}$



299d00b11d853ec14c5375186fa182b68f15a7f2d1fb953b8a36bc6fa85cfcbb

54e20a5a2866ebfae896e34b5251820d7fe31dbb953a4192c5dce5e1c6bcfc22f7e32e6f3fb87b8f6c9ca123
4b358c548d1414c84357254ba212a5f2d4016555

9d9863fe058c560a71b9c169886a86dcc2e2c8425068bd46ece246525af71ae
c0404eec29ce0238346329741f5f1ab73ae46f3246fff55be41a9eef7073cb572

$\left.\begin{matrix} r \\ s \end{matrix}\right\} \sigma$

Till this place

Authenticated EC KAP

A: $u \leftarrow randi$

$A = u * G$

$Sig(PrK_A = z, A) = (r_A, s_A)$

$\xrightarrow{A,\ (r_A, s_A)}$

B: $Ver(PuK_A, (r_A, s_A), A) \to$ Yes

$v \leftarrow randi$

$B = v * G$

$Sig(PrK_B = w, B) = (r_B, s_B)$

$\xleftarrow{B,\ (r_B, s_B)}$

$Ver(PuK_B, (r_B, s_B), B) \to$ Yes

$K_{AB} = u * B = (u \cdot v) * G$ ═══ $K_{BA} = v * A = (v \cdot u) * G$

C:\Users\Eligijus\Documents\Zoom\2021-02-18 18.36.03 Eligijus Sakalauskas's Personal Meeting Room 9999112448

PrK Generation

OpenSSL on-line ??? X X X

Local host 127. . . . . ?? X X

$\Downarrow$

Isolated PC : disconnected from internet

Windows OS [Python Octave . . .]

Formates 'X' → .pem

Generates PuK

Test signat. creation

$x_1 \leftarrow randi(2^{64} - 1)$

x_2

x_3

x_4

$X = x_1 \| x_4 \| x_2 \| x_3$

Flash memory 'X'

Verification with OpenSSL

ECDSA generation and verification with OpenSSL in Windows.

$SHA1(Psw) \to$

$32 + 80 = 112$

$|Psw| = 80 \to 2^{90} \sim 10^{30}$

$|Psw| = 64$ bits $\longrightarrow$ 8 simb. $\}$ $SHA(Psw \| Salt) = h;$
$|Salt| = 32$ bits $\qquad |h| = 160$ bits

$SHA(Psw_1 \| Salt) \stackrel{?}{=} h; \; 2^{64}/2 \longrightarrow 2^{63} \sim 10^{31}$
$Psw_2 \|$ "

8 simb. a - z, A - Z, 0 - 9, , : / .

SHA1( )/sec. $\qquad$ 1000 SHA1/sek $\simeq 2^{10}$ sek.

$$N_{sec} = \frac{2^{63}}{2^{10}} = 2^{53} \sim 10^{28} \text{ sec.}$$

Exponentiating by squaring is a general method for fast computation of large positive integer powers of a number, or more generally of an element of a semigroup, like a polynomial or a square matrix.
Some variants are commonly referred to as **square-and-multiply** algorithms or **binary exponentiation**.
These can be of quite general use, for example in modular arithmetic or powering of matrices.
For semigroups for which additive notation is commonly used, like elliptic curves used in cryptography, this method is also referred to as **double-and-add**.
From <https://en.wikipedia.org/wiki/Exponentiation_by_squaring>

Basic method [edit]

The method is based on the observation that, for a positive integer *n*, we have

$$x^n = \begin{cases} x\,(x^2)^{\frac{n-1}{2}}, & \text{if } n \text{ is odd} \\ (x^2)^{\frac{n}{2}}, & \text{if } n \text{ is even.} \end{cases}$$

This method uses the bits of the exponent to determine which powers are computed.

This example shows how to compute x^{13} using this method. The exponent, 13, is 1101 in binary. The bits are used in left to right order. The exponent has 4 bits, so there are 4 iterations.

First, initialize the result to 1: $r \leftarrow 1\,(= x^0)$.

Step 1) $r \leftarrow r^2\,(= x^0)$; bit 1 = 1, so compute $r \leftarrow r \cdot x\,(= x^1)$;
Step 2) $r \leftarrow r^2\,(= x^2)$; bit 2 = 1, so compute $r \leftarrow r \cdot x\,(= x^3)$;
Step 3) $r \leftarrow r^2\,(= x^6)$; bit 3 = 0, so we are done with this step;
Step 4) $r \leftarrow r^2\,(= x^{12})$; bit 4 = 1, so compute $r \leftarrow r \cdot x\,(= x^{13})$.