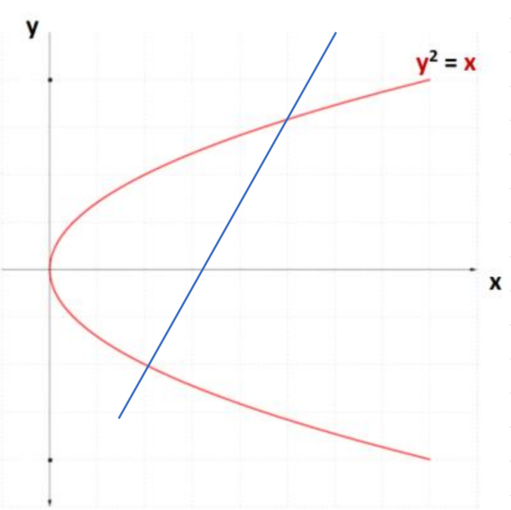
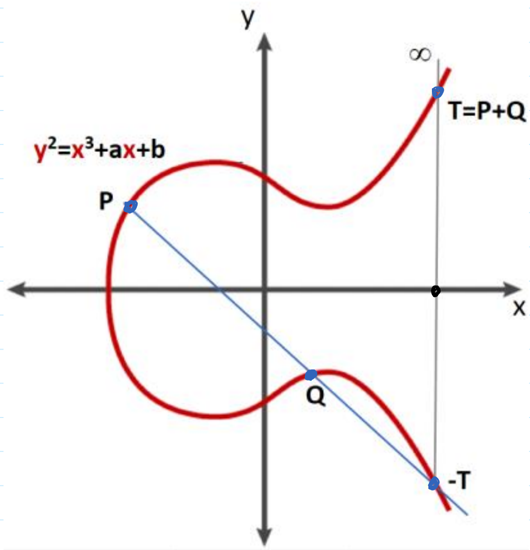
Elliptic Curve Cryptosystem - ECC

In Figures below parabola and elliptic curve (EC) are presented in the plane XOY of real numbers and are expressed by the equations:

$y^2 = x$	$y^2 = x^3 + ax + b$

In EC the point addition operation is defined using two facts:
1. The line crossing any two points in EC intersects with the third point in the curve.
2. The curve is symmetric with respect to axis x since there is y^2 in the left side of EC equation.





$T + (-T) = \infty$
0

The points in EC forms an algebraic additive group with a very special addition operation between points illustrated in EC figure.
Then according to the algebraic group definition the addition of any two points must yield the third point in elliptic curve as a line crossing these two points intersection with the EC.
Question: where line crossing **-*T*** and ***T*** intersects the third point in EC?
Answer: at the infinity.
Paradox: this infinity is named as a zero of EC group since any additive group must have a neutral element called zero: $T + (-T) = 0$, and $T + 0 = T$.

Finite Field is denoted by F_p (or rarely Z_p), when p is prime.
$F_p = \{0, 1, 2, 3, \ldots, p-1\}$; where addition, multiplication, substraction and division operations are performed **mod *p***: $+_{\text{mod } p}$, $-_{\text{mod } p}$, $\bullet_{\text{mod } p}$, $:_{\text{mod } p}$.
Cyclic Group: $Z_p^* = \{1, 2, 3, \ldots, p-1\}$; $\bullet_{\text{mod } p}$, $:_{\text{mod } p}$.

Let us consider abstract EC defined in the plane XOY with coordinates in finite field and $F_p = \{0, 1, 2, \ldots, p-1\}$ and expressed by the equation:

$$y^2 = x^3 + ax + b \text{ mod } p.$$

EC points are computed by choosing coordinate x and computing coordinate y^2.

To compute coordinate ***y*** it is needed to extract root square of y^2.

$$y = \pm\sqrt{y^2} \bmod p.$$

Notice that from y^2 we obtain 2 points in EC, namely ***y*** and -***y*** no matter computations are performed with integers **mod *p*** or with real numbers.
Notice also that since EC is symmetric with respect to ***x***-axis, the points ***y*** and -***y*** are symmetric in EC.
Since all arithmetic operations are computed **mod *p*** then according to the definition of negative points in F_p points ***y*** and -***y*** must satisfy the condition

$$y + (-y) = 0 \mod p.$$

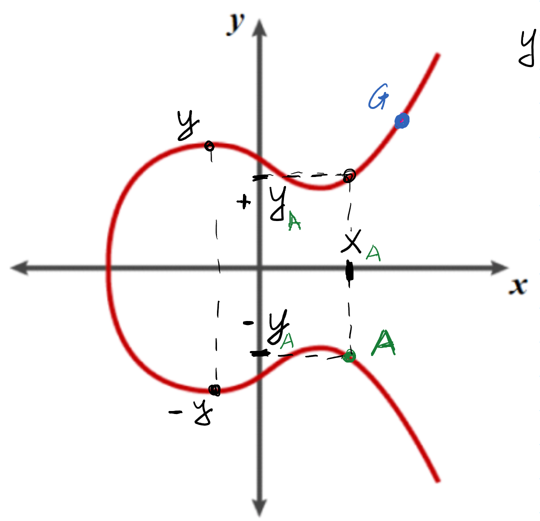
Then evidently

$$y^2 = (-y)^2 \bmod p.$$

For example: ***p*** = 11
-2 mod 11 = 9
2^2 mod 11 = 4 & 9^2 mod 11 = 4

```
>> mod(9^2,11)
ans = 4
```

The positive and negative coordinates ***y*** and -***y*** in EC in the real numbers plane XOY are presented in Fig.
The positive and negative numbers for ***p***=11 are presented in table .



$y - y = 0$

$1 - 1 = 0$

$|p| = 256$ b.

$|x_A| = 256$ b,

$|y_A| = 256$ b.

y mod 11			(-***y***) mod 11
1	odd	even	-1=10
2	even	odd	-2=9
3	odd	even	-3=8
4	even	odd	-4=7
5	odd	even	-5=6
6	even	odd	-6=5
7	odd	even	-7=4
8	even	odd	-8=3
9	odd	even	-9=2
10	even	odd	-10=1

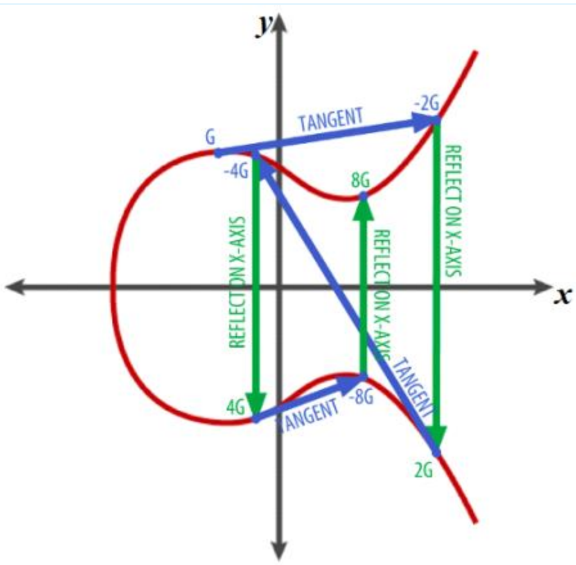
$x_A \rightarrow$ tašho A koord y_A.

Notice that performing operations **mod *p*** if ***y*** is odd then -***y*** is and vice versa.

ElGamal Cryptosystem (CS)	**Elliptic Curve Cryptosystem (CS)**
PP=(strongprime ***p***, generator ***g***); ***p***=255996887; ***g***=22;	**PP**=(EC **secp256k**; BasePoint-Generator ***G***; prime ***p***; param. ***a***, ***b***); Parameters ***a***, ***b*** defines EC equation $y^2=x^3+ax+b$ **mod *p*** over F_p.
PrK=***x***; >> **x**=randi(***p***-1).	$\mathbf{PrK_{ECC}}$=**z**; >> **z**=randi(***p***-1). $\lvert z \rvert = 256$ b, since $\lvert p \rvert = 256$b.
PuK=***a***=g^x **mod *p***.	$\mathbf{PuK_{ECC}}$=***A***=**z******G***. $\lvert A \rvert = \lvert (x_A, y_A) \rvert = 256 + 256 = 512$b.
Alice *A*: **x**=**1975596**; ***a***=**210649132**;	**Alice *A***: **z**=**.....**; ***A***=(x_A, y_A);

Alice A: x=**1975596**; a=**210649132**; **Alice** A: z=**…..**; $A=(x_A, y_A)$;

This property allows us to reduce bit representation of **PuK$_{ECC}$**$=A=z*G=(x_A, y_A)$;
In normal representation of **PuK$_{ECC}$** it is needed to store 2 coordinates (x_A, y_A) every of them having 256 bits. For **PuK$_{ECC}$** it is required to assign 512 bits in total.
Instead of that we can store only x_A coordinate with an additional information either coordinate y_A is odd or even.
The even coordinate y_A is encided by prefix **02** and odd coordinate y_A is encoded by prefix **03**.
It is a compressed form of **PuK$_{ECC}$**.
If **PuK$_{ECC}$** is presented in uncompressed form than it is encoded by prefix **04**.
Imagine, for example, that having generator G we are computing **PuK$_{ECC}$**$=A=z*G=(x_A, y_A)$ when $z=8$.
Please ignore that after this explanation since it is crasy to use such a small z. It is a gift for adversary To provide a search procedure.
Then **PuK$_{ECC}$** is represented by point $8G$ as depicted in Fig. So we obtain a concrete point in EC being either even or odd.
The coordinate y_A of this point can be computed by having only coordinate x_A using formulas presented above and having prefix either **02** or **03**.



EC: $y^2=x^3+ax+b \mod p$
Let we computed **PuK$_{ECC}$**$=A=(x_A, y_A)=8G$.
Then $(y_A)^2=(x_A)^3+a(x_A)+b \mod p$ is computed.
By extracting square root from $(y_A)^2$ we obtain 2 points:
$8G$ and $-8G$ with coordinates (x_A, y_A) and$(x_A, -y_A)$.
According to the property of arithmetics of integers **mod** p either y_A is **even** and $-y_A$ is **odd** or y_A is **odd** and $-y_A$ is **even**.
The reason is that $y_A+(-y_A)=0 \mod p$ as in the example above when $p=11$.
Then we can compress **PuK$_{ECC}$** representation with 2 coordinates (x_A, y_A) by representing it with 1coordinate x_A and adding prefix either 02 if y_A is even or 03 if y_A is odd.

$2^{3} = 8$ sums

Because this curve is defined over a finite field of prime order instead of over the real numbers, it looks like a pattern of dots scattered in two dimensions, which makes it difficult to visualize. However, the math is identical to that of an elliptic curve over real numbers. As an example, Elliptic curve cryptography: visualizing an elliptic curve over F(p), with p=17 shows the same elliptic curve over a much smaller finite field of prime order 17, showing a pattern of dots on a grid. The secp256k1 bitcoin elliptic curve can be thought of as a much more complex pattern of dots on a unfathomably large grid.

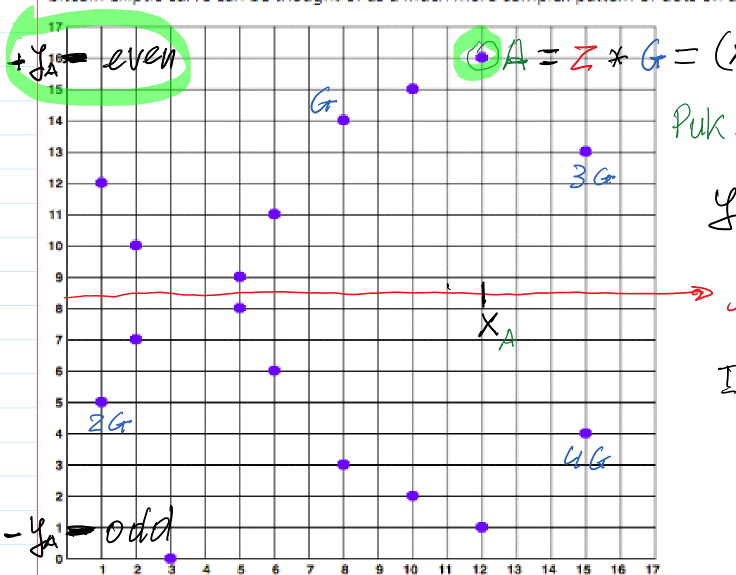


Figure 3. Elliptic curve cryptography: visualizing an elliptic curve over F(p), with p=17

$A = z * G = (x_A, y_A)$

$PuK = A = 03\ x_A := 03\ 0x\ \underline{64 \text{ hex numb.}}$

$y_A^2 = x_A^3 + a(x_A) + b$

$y_A = \pm\sqrt{y_A^2}$

If y_A even → accept

$-y_A$ even → accept

$n = p.$

v, r, s, *Ethereum signature* r, s, v

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.

Packages	2021.12.05 18:23	Python File	1 KB
ECC	2021.12.09 19:06	Python File	9 KB

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**, when **n** is the number of EC points.

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.

Ethereum signature creation for message ***M***

Signature is formed on the h-value ***h*** of Hash function of ***M***.

Recommended to use **keccak256** algorithm of 256 bit length of h-value.

h = H(***M***)=keccak256(***M***);

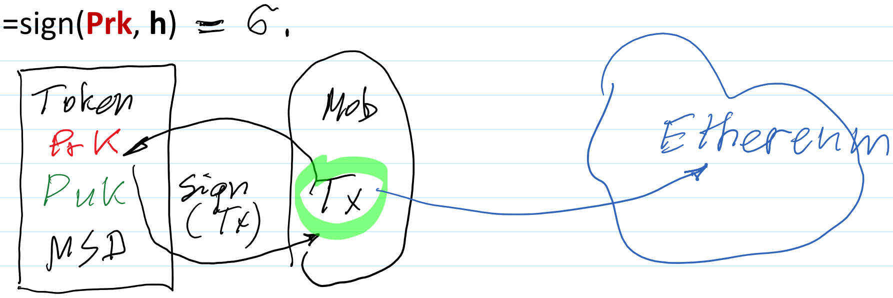
Signing and Verifying Ethereum Signatures – Yos Riady · Software Craftsman

You can sign messages entirely off-chain in the browser, without interacting with the Ethereum network. Signing and the verification of ECDSA-signed messages allows tamper proof communications outside of the blockchain.
We can call the via an Ethereum eth_sign method client such as web3.js:

```
// Create a SHA3 hash of the message 'Apples'
const messageHash = web3.sha3('Apples');
// Signs the messageHash with a given account
const signature = await
web3.eth.personal.sign(messageHash,
web3.eth.defaultAccount);
```

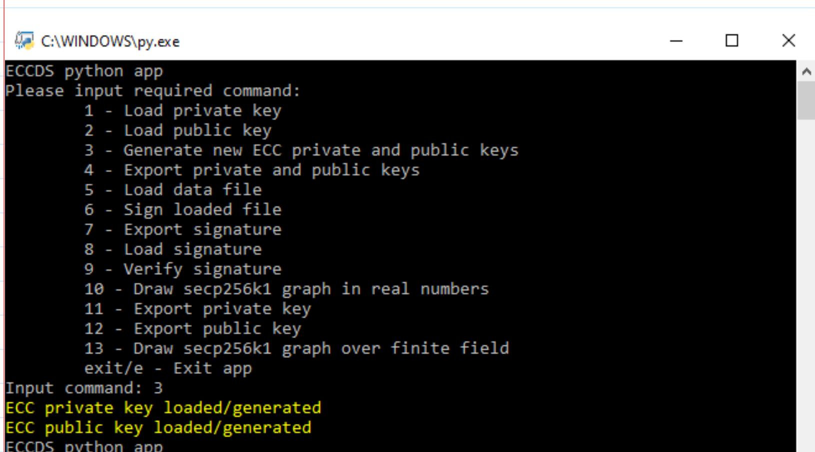
The **eth_sign** method calculates an Ethereum specific signature with:
eth_sign(keccak256("\x19Ethereum Signed Message:\n" + len(message) + message))).
The prefix to the message makes the calculated signature recognisable as an Ethereum specific signature.>

sigma=sign(**Prk**, **h**) $= \sigma$.

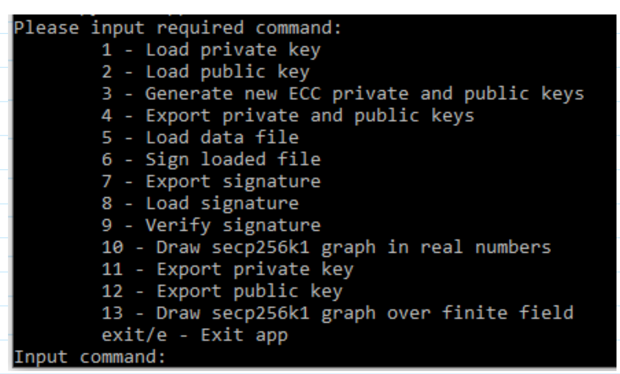


Iš praeito pusmečio siuntimo, kam galėtų reikėti patalpinau į we transfer: https://we.tl/t-V0FlMXQ2fz.

111.ECDSA-Python http://crypto.fmf.ktu.lt/xdownload/



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



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

App_PrK	2023.02.04 12:56	Text Document	1 KB
App_PuK	2023.02.04 12:56	Text Document	1 KB

0xb20bcc56bebccc51557e2d4b32e1c99604dca974b05df7c91d057f8d202770d1 $z = PrK$

2d9c5a458b25fc28e9f0591f4dc397982130aee2844af82bc5e6108608246a0f x_A
36095891fe2c96bb1a429794f56a886b125ab29c0f381826beb y_A $\} A = PuK$

11

17 BC 3
A

03 0x2d9c5a458b25fc28e9f0591f4dc397982130aee2844af82bc5e6108608246a0f X_A

0x686a0f209c1bc05617f8b540afbd5a49651453ee48b472256e6537d2b9684513 **PrK = *z***

fba40304078e170874e204afd87ff6d9f15f0c58b81a60e59e0a3104063c0667 **$PuK_x = x_A$**
58183c3388ddf828e81e23f1d3265ef69f52d39a837eed9d91e4c2d17d15f8c1 **$PuK_y = y_A$**
1 is odd number, prefix=03

03 0xfba40304078e170874e204afd87ff6d9f15f0c58b81a60e59e0a3104063c0667

Private Key of EC Cryptosystem (ECC) is **PrK** $_{ECC}$=*z*, where *z* is secret integer generated at random, i.e. *z* ← randi.
Public Key of ECC is **PuK_{ECC}**=A=z*G=(x_A, y_A),
where * means generator G multiplication by integer z or this means z-times addition of point G in EC according to points addition rule defined above in Fig.

Signature creation for message ***M***

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

1. $h = \mathrm{H}(M) = \mathrm{SHA256}(M)$;
2. i <-- randi; $|i| \leq 256$ bits;
3. $R = i * G = i * (x_G, y_G) = (x_R, y_R)$;
4. $r = x_R \bmod p$;
5. $s = (h + z \bullet r) \bullet i^{-1} \bmod p$; $|s| \leq 256$ bits; // Since p is prime, then exists $i^{-1} \bmod p$.
   // >> i_m1=mulinv(i,p) % in Octave σ
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 $V = u_1 * G + u_2 * A = V(x_V, y_V)$
3. The signature is valid if $R=V$; $r = x_V = x_R \bmod p$.

ECDSA	ElGamal Signature	Schnorr Signature
$h = \mathrm{H}(m)$;	$h = \mathrm{H}(m)$;	$h = \mathrm{H}(m)$;
$i \leftarrow$ randi; Compute $i^{-1} \bmod p$	$i \leftarrow$ randi; $\gcd(i, p-1)=1$ Compute $i^{-1} \bmod (p-1)$	$i \leftarrow$ randi;
$R = i * G = i * (x_G, y_G) = (x_R, y_R)$; $r = x_R \bmod p$; $|i| \leq 256$ bits;	$r = g^i \bmod p$;	$r = g^i \bmod p$;
$s = (h + z \bullet r) i^{-1} \bmod p$; $|s| \leq 256$ bits;	$s = (h - x \bullet r) i^{-1} \bmod (p-1)$;	$s = (i + x \bullet h) \bmod (p-1)$;
$s^{-1} = (h + z \bullet r)^{-1} i \bmod p$;	$h = xr + is \bmod (p-1)$.	
Sign(**PrK$_{ECC}$=z**, h) = $(r, s) = \sigma$;	**Sign**(**PrK=x**, h) = $(r, s) = \sigma$;	**Sign**(**PrK=x**, h) = $(r, s) = \sigma$;
ECDSA Verification	**ElGamal Signature Verification**	**Schnorr Signature Verification**
Compute $u_1 = h \bullet s^{-1} \bmod p$ and $u_2 = r \bullet s^{-1} \bmod p$;	Compute: $u_1 = g^h \bmod p$; and $u_2 = a^r r^s \bmod p$	Compute: $u_1 = g^s \bmod p$. and $u_2 = r a^h \bmod p$
Compute $R = u_1 * G + u_2 * A = (x_R, y_R)$;	Signature is valid if: $u_1 = u_2$	Signature is valid if: $u_1 = u_2$
The signature is valid if $r = x_R \bmod p$.		

Let u, v are integers < p.
Property 1: $(u + v) * P = u * P \boxplus v * P$ replacement to --> $\underline{(u + v)P = uP + vP}$
Property 2: $(u) * (P \boxplus Q) = u * P \boxplus u * Q$ replacement to --> $u(P + Q) = uP + uQ$

Important identity used e.g. in Ring Signature:
(t-zc)*G+c*A = t*G-zc*G+c*A = t*G-c(z*G)+c*A = t*G-c*A+c*A = tG mod p.

$u + v * P$ $\sqrt{a+x} - \sqrt{a} = \sqrt{x}$

Correctness:
$R = u_1 * G + u_2 * A$
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:

$\boldsymbol{R}=(u_1 + u_2 \bullet \mathbf{z}) * \boldsymbol{G}$

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

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

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

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

Expanding the definition of ***s*** from signature creation we have:

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

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 $\boldsymbol{R} = \boldsymbol{i} * \boldsymbol{G} = (\boldsymbol{x_R}, \boldsymbol{y_R})$; $\mathbf{r} = \mathbf{x_R}$.

$\mathbf{PrK}_{\mathbf{ECC}} = \mathbf{z} < \mathbf{n} < 2^{256}$; $\mathbf{PuK}_{\mathbf{ECC}} = \boldsymbol{A} = (\boldsymbol{a_x}, \boldsymbol{a_y})$;

$|\mathbf{PrK}_{\mathbf{ECC}} = \mathbf{z}| = 256$ bits; $|\mathbf{PuK}_{\mathbf{ECC}} = \mathbf{A}| = 512$ bits.

Doubling points in EC

$\boldsymbol{A} = \mathbf{11} * \boldsymbol{G}$

$\mathbf{11 = 1011_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 8 + 2 + 1 = 11.}$

$\mathbf{11 = 1011_2 = 2 \cdot 2 \cdot 2 + 0 \cdot 2 \cdot 2 + 1 \cdot 2 + 1 = 2 \cdot 2 \cdot 2 + 2 + 1}$ // $* \boldsymbol{G}$

$\boldsymbol{A} = \mathbf{2*(2*(2*}\boldsymbol{G}\mathbf{))} \boxplus \mathbf{0*}\boldsymbol{G} \boxplus \mathbf{2*}\boldsymbol{G} \boxplus \mathbf{1*}\boldsymbol{G}$

$\boldsymbol{A} = \mathbf{(8*}\boldsymbol{G}\mathbf{)} \boxplus \mathbf{2*}\boldsymbol{G} \boxplus \boldsymbol{G}.$

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

IBM
Peter Shorr
433 qbits
quantum
entalgement
$1KB = 1024$
1024^2
2^{1024}

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} birthday paradox $2^{256} = \sqrt{2^{512}}$

2^{112} secure against brute force attack