Elliptic Curve Cryptography (ECC)

Elliptic Curve Cryptography (ECC) leverages algebraic structure of elliptic curves over finite fields in order to generate public keys. It can solve the problem of performance-heavy operations with too large keys (f.ex. 2048-bit RSA), as usually for ECC 256 bits are used.

2 NIST elliptic curves

2.1 NIST Standard

National Institute of Standards and Technology (NIST) is an American national institute, working under the provisions of the Federal Information Security Management Act. It defines the standars of the curves and its parameters for different key-lengths, in order to achieve implementation with base security accepted by the USA government. As well, it outlines key-establishment schemes, key-derivation and key-pair managements methods, as well as recommendations for DLC-based (Discrete Logarithm Cryptography) key-agreement schemes[4].

For each key-length NIST provides two types of fields[5]:

- **prime field**: it contains a prime number p of elements, which are the integers modulo p, with the field arithmetic being implemented in terms of the arithmetic of integers modulo p;
- **binary field**: it contains 2^m elements (with m being the degree of the field) which are bit strings of length m, and the field arithmetic is implemented in terms of operations on bits.

As well, two types of curves are given:

- **pseudorandom** curves: with coefficients that are generated from the output of the seeded hash function;
- **special** curves: with coefficients and the field being selected to optimize the efficiency of the arithmetic operations on the curve.

The special curves over prime fields are known as *Edwards* or *Montgomery* curves; meanwhile over binary field they are referred as *Koblitz* curves.

In this paper, there will be inspected in detail psudorandom curves over prime fields.

2.2 NIST pseudorandom curves over prime fields

2.2.1 Curves parameters

Due to NIST official publication on *Recommended Elliptic Curves for Federal Government Use*[6], all the pseudorandom curves over prime field curves are defined by:

$$E : y^2 = x^3 - 3x + b \mod p$$

with a having a static value of $a = -3$ in every case for the reasons of efficiency.

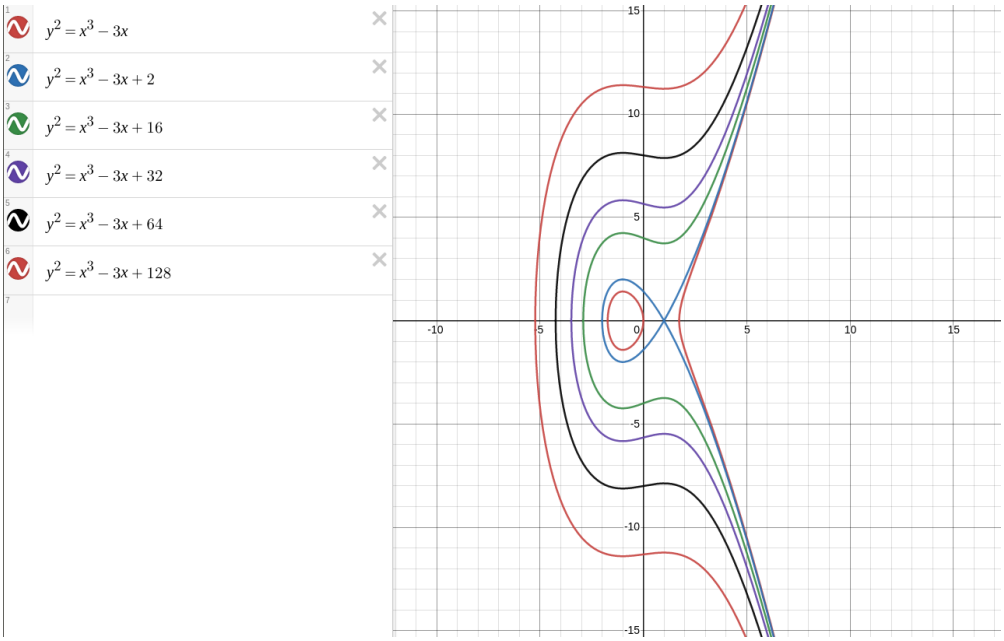


Fig. 2.1: Examples of curve E with different coefficients b, interactive access: desmos tool

The following parameters are specifying the curves in details:

- p: prime modulus
- n: the order
- $SEED$: the 160-bit long input seed to the domain parameter generation (SHA-1 algorithm)
- c: the output of the SHA-1 algorithm
- b: the coefficient of the curve, that grants satisfaction of condition $b^2c = -27 \mod p$
- G: the base point with two values:
  - Gx: the x coordinate of the base point
  - Gy: the y coordinate of the base point

The parameters p and n are always given in decimal value, meanwhile the rest in hexadecimal.

2.2.2 P-256: 256-bit length curve

	param	value
	$p_{(10)}$	115792089210356248762697446949407573530086143415290314195533631308867097853951
	$n_{(10)}$	115792089210356248762697446949407573529996955224135760342422259061068512044369
$SEED_{(16)}$		c49d3608 86e70493 6a6678e1 139d26b7 819f7e90
	$c_{(16)}$	7efba166 2985be94 03cb055c 75d4f7e0 ce8d84a9 c5114abc af317768 0104fa0d
	$b_{(16)}$	5ac635d8 aa3a93e7 b3ebbd55 769886bc 651d06b0 cc53b0f6 3bce3c3e 27d2604b
$G_{x(16)}$		6b17d1f2 e12c4247 f8bce6e5 63a440f2 77037d81 2deb33a0 f4a13945 d898c296
$G_{y(16)}$		4fe342e2 fe1a7f9b 8ee7eb4a 7c0f9e16 2bce3357 6b315ece cbb64068 37bf51f5

2.2.3 P-521: 512-bit length curve

param	value
$p_{(10)}$	6864797660130609714981900799081393217269435300143305409394463459185543183397656052122559640661454554977296311
$n_{(10)}$	6864797660130609714981900799081393217269435300143305409394463459185543183397655394245057746333217197532963995
$SEED_{(16)}$	d09e8800 291cb853 96cc6717 393284aa a0da64ba
$c_{(16)}$	0b4 8bfa5f42 0a349495 39d2bdfc 264eeeeb 077688e4 4fbf0ad8 f6d0edb3 7bd6b533 28100051 8e19f1b9 ffbe0fe9 ed8a3c22 00b8f875
$b_{(16)}$	051 953eb961 8e1c9a1f 929a21a0 b68540ee a2da725b 99b315f3 b8b48991 8ef109e1 56193951 ec7e937b 1652c0bd 3bb1bf07 3573
$Gx_{(16)}$	c6 858e06b7 0404e9cd 9e3ecb66 2395b442 9c648139 053fb521 f828af60 6b4d3dba a14b5e77 efe75928 fe1dc127 a2ffa8de 3348b3c
$Gy_{(16)}$	118 39296a78 9a3bc004 5c8a5fb4 2c7d1bd9 98f54449 579b4468 17afbd17 273e662c 97ee7299 5ef42640 c550b901 3fad0761 353c7

1 Elliptic curves

1.1 Elliptic curve: definition

Elliptic curves are defined over a field K and describe points in its cartesian product K^2; they are smooth and projective affine algebraic structures.

The Weierstrass equation of an elliptic curve E over a field K is [1]:

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

what in case of the fields with characteristic different than 2 or 3 (i.e. the smallest number of summands n of the field's identity element e so $ne = 0$), the affinitic equation of the elliptic curve can be simplified to:

$$E : y^2 = x^3 + Ax + B$$

$$A, B \in K$$

1.2 Elliptic curve: point addition

The structure of elliptic curves fulfills the necessary requirements for its points to form an algebraic group with a certain addition law.

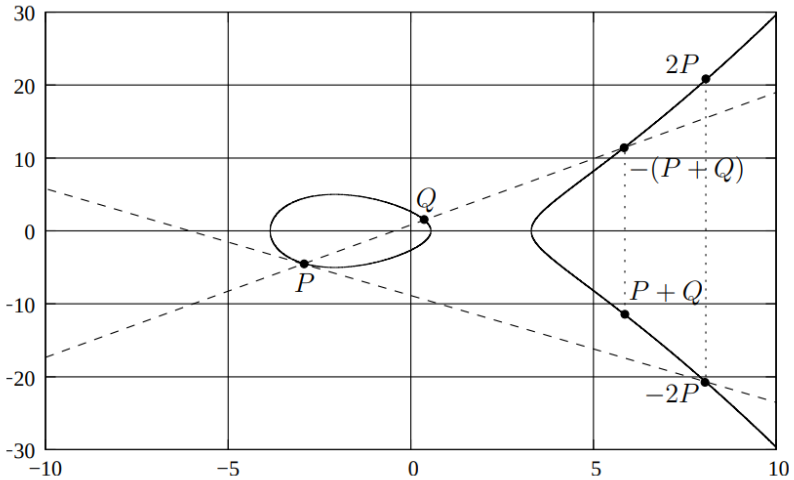


Fig. 1.1: Operations on en elliptic curve [[2]](#references)

With $P, Q \in E$ - being two points on an elliptic curve group E - and L being the line connecting those points, the third point R can be defined where the intersection of L with E happens. Then, the result of $P + Q$ is located on the residual point of the intersection R'.

The coordinates of the result (x_{res}, y_{res}) can be calculated as:

$$x_{res} = s^2 - x_p - x_q$$

$$y_{res} = -y_p + s(x_p - x_{res})$$

with s being the slope of the line L through P and Q:

$$s = y_p - y_q / x_p - x_q$$

With O being the neutral point in the group E, the following properties are fullfiled:

- $P + O = P$
- $P + Q = Q + P$
- if $P \in E$ then there exists a point Q such that $P + Q = O$
- if $P, Q \in E$ then $P + Q \in E$
- for $P, Q, R \in E$ is satisfied $(P + Q) + R = P + (Q + R)$

1.4 Multiplication of EC point with integer

Any point P on the elliptic curve can be added to itself resulting in:

$$P + P = 2 * P$$

Continuing the addition multiple n times yeilds to:

$$P * n = P + P + \ldots + P(n - times)$$

As multiplication is defined by the point addition, it always results in another point on the elliptic curve.

For point doubling, i.e. $R = 2 * P = P + P$, the following formula holds:

$$x_R = s^2 - 2x_p$$

$$y_R = -y_p + s(x_p - x_{res})$$

with s:

$$s = (3x_p^2 + a)/2y_p$$

1.5 Elliptic Curves over finite field

The ECs used in ECC are strictly defined over finite field Fp, with p being a prime number bigger than 3. Hence, the field is a square matrix with dimensions $p \times p$. Hence, the points of a given curve are limited to integer coordinates within its field matrix and all algebraic operations will result in a point within that field.

3 Brainpool elliptic curves

3.1 Brainpool standard

Brainpool elliptic curves are curves which are standardized by the Internet Engineering Task Force (IETF) - an international open standards organization - and are described in RFC-5639[8].

It presents a set of domain parameters defining cryptographically strong groups on elliptic curves for different key-length encryption schemes.

The curves are described with regard to Wierstrass' definition:

$$E : y^2 = x^3 + Ax + B$$

3.2 Brainpool curves

3.3.1 Curves parameters

The Brainpool curves are defined with similar parameters to NIST standard; used in the official publication symbols are:

- p: prime number (modulus) specifying the base field, $p > 3$
- a and b: the coefficients of the E, satisfying $4a^3 + 27b^2 \mod p \neq 0$
- G: the base point with two values:
  - x: the x coordinate of the base point
  - y: the y coordinate of the base point
- q: the prime order of the group generated by base point G (equivalent of NIST n parameter), such that $qG = 0$
- h: the cofactor of G in E, $h = E(\mathrm{Fp})/q$

In the further implementations, the given here parameters naming convention will be used.

3.3.2 brainpoolP256r1: 256-bit length curve

The curve for 256-bit length key is identified by the Curve-ID `brainpoolP256r1`, and has the following parameters:

param	value

	param	value
	$p_{(16)}$	A9FB57DBA1EEA9BC3E660A909D838D726E3BF623D52620282013481D1F6E5377
	$q_{(16)}$	A9FB57DBA1EEA9BC3E660A909D838D718C397AA3B561A6F7901E0E82974856A7
	$a_{(16)}$	7D5A0975FC2C3057EEF67530417AFFE7FB8055C126DC5C6CE94A4B44F330B5D9
	$b_{(16)}$	26DC5C6CE94A4B44F330B5D9BBD77CBF958416295CF7E1CE6BCCDC18FF8C07B6
$Gx_{(16)}$		8BD2AEB9CB7E57CB2C4B482FFC81B7AFB9DE27E1E3BD23C23A4453BD9ACE3262
$Gy_{(16)}$		547EF835C3DAC4FD97F8461A14611DC9C27745132DED8E545C1D54C72F046997
	$h_{(10)}$	1

3.3.3 brainpoolP512r1: 512-bit length curve

The curve for 512-bit length key is identified by the Curve-ID `brainpoolP512r1`, and has the following parameters:

param	value
$p_{(16)}$	AADD9DB8DBE9C48B3FD4E6AE33C9FC07CB308DB3B3C9D20ED6639CCA703308717D4D9B009BC66842AECDA12AE6A380E62881
$q_{(16)}$	AADD9DB8DBE9C48B3FD4E6AE33C9FC07CB308DB3B3C9D20ED6639CCA70330870553E5C414CA92619418661197FAC10471DB1D3
$a_{(16)}$	7830A3318B603B89E2327145AC234CC594CBDD8D3DF91610A83441CAEA9863BC2DED5D5AA8253AA10A2EF1C98B9AC8B57F1117
$b_{(16)}$	3DF91610A83441CAEA9863BC2DED5D5AA8253AA10A2EF1C98B9AC8B57F1117A72BF2C7B9E7C1AC4D77FC94CADC083E6798405(
$Gx_{(16)}$	81AEE4BDD82ED9645A21322E9C4C6A9385ED9F70B5D916C1B43B62EEF4D0098EFF3B1F78E2D0D48D50D1687B93B97D5F7C6D50
$Gy_{(16)}$	7DDE385D566332ECC0EABFA9CF7822FDF209F70024A57B1AA000C55B881F8111B2DCDE494A5F485E5BCA4BD88A2763AED1CA2E
$h_{(10)}$	1

4 Implementation: Elliptic curve over finite prime field and ECDH

4.1 Preeliminary considerations

In order to achieve certain level of security, the selection of the curve parameters and the base point is important; hence, in this realization, the Brainpool defined curves will be used.

Apart of that, the required protection depends on the implementation and its environment: if the hardware that runs it is completely secured and separated, the time-invariant implementation may be sufficient. If the implementation of ECC is located on the physically accessible device, it has to be robust against side-channel attacks (power consumption and its analysis, electromagnetic emanations, active fault injection, etc.), then more security measures needs to be taken, with techniques of randomization being essential in it[3]. In this lab, the hardware vurnerabilities are not taken in considerations, hence the focus will be on achieving time-constant implementation.

Furthermore, all the processing will be performed on a personal machine. This has to be taken in consideration when evaluating time measurements of the processes execution. Below are presented the specifications of the processor.

In [5]:
```
%run systemInfo.py
```

```
[+] Architecture : 64bit
[+] System Name : Linux
[+] Operating System Version : #101~18.04.1-Ubuntu SMP Fri Oct 22 09:25:04 UTC 2021
[+] Platform : Linux-5.4.0-90-generic-x86_64-with-glibc2.27
[+] Processor : x86_64
[+] System Uptime : 0:27 hours
[+] Total number of processes : 357
[+] Number of Physical cores : 4
[+] Number of Total cores : 8
[+] Max Frequency : 4900.00Mhz
[+] Min Frequency : 400.00Mhz
[+] Current Frequency : 1190.73Mhz
[+] Total CPU Usage : 13.9%
[+] Processor 0 :  Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz
[+] Processor 1 :  Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz
```

```
[+] Processor 2 :  Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz
[+] Processor 3 :  Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz
[+] Processor 4 :  Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz
[+] Processor 5 :  Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz
[+] Processor 6 :  Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz
[+] Processor 7 :  Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz
[+] Total Memory present : 15.34 Gb
[+] Total Memory Available : 12.02 Gb
[+] Total Memory Used : 2.51 Gb
[+] Percentage Used : 21.7 %
[+] Total swap memory : 32.0
[+] Free swap memory : 32.0
[+] Used swap memory : 0.0
[+] Percentage Used : 0.0%
```

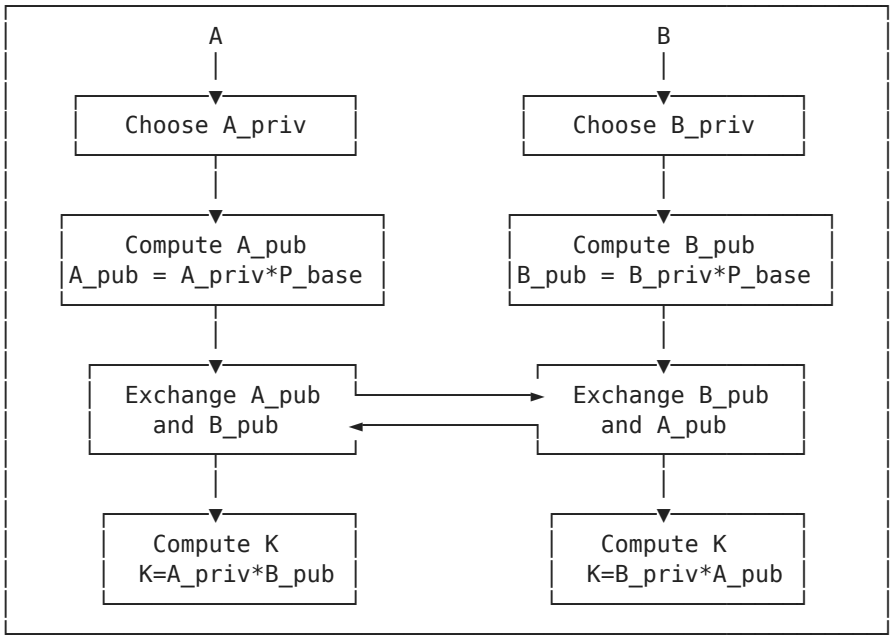
4.2 ECDH: Elliptic Curve Diffie-Hellman Key Exchange

The ECDH (Elliptic Curve Diffie-Hellman Key Exchange) is the scheme that allows derivation of the shared secret key over an insecure channel, based on the private-public key pairs of both sides. It follows the traditional Diffie–Hellman Key Exchange, nevertheless, the key pairs are of elliptic curve type and point multiplication is performed instead of the modular exponentiation.

Considering following properties:

- P_{base} - base point of the curve
- A_{priv} - integer, private key of A
- A_{pub} - point on the curve, $A_{pub} = A * P_{base}$, public key of A
- B_{priv} - integer, private key of B
- B_{pub} - point on the curve, $B_{pub} = B * P_{base}$, public key of B
- K - shared secret key, $K = A_{priv} * B_{priv} * P_{base}$

the algorithm executes following flow:



This results in both sides, A and B, having the same secret shared key that can be used for symmetric encryption.

Based on that, the implementation will be divided into following steps:

- curve data type definition
- curve and field generation functions
- point addition function
- double-and-add algorithm for faster multiplication
- Diffie-Hellman algorithm for shared key generation

4.3 Definition of the curve parameters

The curve parameters were declared with use of Python's `dataclass` [7], introduced in version 3.7. This allowed to keep the code cleaner and to approach further computations rather functionally than object oriented.

In [6]:
```
%reset
```

In [7]:
```
from dataclasses import dataclass, field
from typing import Callable, Any

@dataclass
class Point:
    x: int
    y: int
    def __eq__(self, other):
        return self.x == other.x and self.y == other.y
    def __neq__(self, other):
        return not self == other

@dataclass
class Field:
    p: int # prime modulus
    q: int # order of the base point
    h: int = 1

@dataclass
class EllipticCurve:
    a: int # coefficient
    b: int # coefficient
    base: Point # base point
    field: Field
    inf: Point = Point(0, 0) # point of infinity
    name: str = "undefined"
```

A list of elliptic curves registry was created. In order to extend it, the curve's name needs to be specified in the `ELLIPTIC_CURVES` enum, followingly the object should be instantiated as `EllipticCurve(...args)` and then added to a list `ELLIPTIC_CURVES_LIST`. Each curve can be later retrived with the function `getCurve(name)` where name is a key or a valid value of `ELLIPTIC_CURVES` enum. F.ex.:

```
ec1 = getCurve(ELLIPTIC_CURVES.bp512.value)
ec2 = getCurve('brainpoolP256r1')
```

TODO: automatize it more and read parameters of each curve from json file.

In [8]:
```
from enum import Enum

# list of available curves
class ELLIPTIC_CURVES(Enum):
    bp256 = 'brainpoolP256r1'
    bp512 = 'brainpoolP512r1'

brainpool_256 = EllipticCurve(
    name=ELLIPTIC_CURVES.bp256.value,
    a=0x7D5A0975FC2C3057EEF67530417AFFE7FB8055C126DC5C6CE94A4B44F330B5D9,
    b=0x26DC5C6CE94A4B44F330B5D9BBD77CBF958416295CF7E1CE6BCCDC18FF8C07B6,
    base=Point(
        x=0x8BD2AEB9CB7E57CB2C4B482FFC81B7AFB9DE27E1E3BD23C23A4453BD9ACE3262,
        y=0x547EF835C3DAC4FD97F8461A14611DC9C27745132DED8E545C1D54C72F046997
    ),
    field=Field(
        p=0xA9FB57DBA1EEA9BC3E660A909D838D726E3BF623D52620282013481D1F6E5377,
        q=0xA9FB57DBA1EEA9BC3E660A909D838D718C397AA3B561A6F7901E0E82974856A7,
        h=1
    )
)

brainpool_512 = EllipticCurve(
    name=ELLIPTIC_CURVES.bp512.value,
    a=0x7830A3318B603B89E2327145AC234CC594CBDD8D3DF91610A83441CAEA9863BC2DED5D5AA8253AA10A2EF1C98B9A
    b=0x3DF91610A83441CAEA9863BC2DED5D5AA8253AA10A2EF1C98B9AC8B57F1117A72BF2C7B9E7C1AC4D77FC94CADC08
    base=Point(
        x=0x81AEE4BDD82ED9645A21322E9C4C6A9385ED9F70B5D916C1B43B62EEF4D0098EFF3B1F78E2D0D48D50D1687E
```

```
        y=0x7DDE385D566332ECC0EABFA9CF7822FDF209F70024A57B1AA000C55B881F8111B2DCDE494A5F485E5BCA4BD8
    ),
    field=Field(
        p=0xAADD9DB8DBE9C48B3FD4E6AE33C9FC07CB308DB3B3C9D20ED6639CCA703308717D4D9B009BC66842AECDA12A
        q=0xAADD9DB8DBE9C48B3FD4E6AE33C9FC07CB308DB3B3C9D20ED6639CCA70330870553E5C414CA92619418661197
        h=1
    )
)

ELLIPTIC_CURVES_LIST = [brainpool_256, brainpool_512]

# function to retrieve a curve to work on
getCurve: Callable[[ELLIPTIC_CURVES], EllipticCurve] = lambda name : next(filter(lambda ec: ec.name
```

In [9]:

```
print(getCurve(ELLIPTIC_CURVES.bp256.value))
print(getCurve(ELLIPTIC_CURVES.bp512.value))
print(getCurve('brainpoolP256r1'))
```

```
EllipticCurve(a=56698187605326110043627228396178346077120614539475214109386828188763884139993, b=175
77232497321838841075697789794520262950426058923084567046852300633325438902, base=Point(x=63243729749
56233335529224355031297033477817557105472658709538162362714411478 6, y=3821861509375352389312227796 40
30810387585405539772602581557831887485717997975), field=Field(p=76884956397045344220809746629001649092737531784414529 53
87555190630635363590 79, h=1), inf=Point(x=0, y=0), name='brainpoolP256r1')
EllipticCurve(a=62948605579730632276664213064763793240747157706227462271369104454503019142812760980 2
79909684079839626911518536785638778342218340274397182380657258442641 38, b=32457890083289670592748495
84342077916531909009637501918328323668736179176583263496463525128488282611559800773506973771797764 81
14988349952343415308622866 27, base=Point(x=67920591404245751744356404312691950878431533901025218814 6
80230127320474825798530775456474462728667949363715224107745326865824846179460139288742968443515 22, y
=65922445552401128733247483814296103413127129403262663313274450666870105454152564610977074832886502 1
69926130901850429577163183011801592347885043076285093 30), field=Field(p=89489622076502325516560 2815
15915342216260964409835451134459718720005701041355243991793430419195694276544653038642734593796389 43
09923928536070534607816947, q=89489622076502325516566028151591534221626096440983545113445971872000 57
01041341852837898173064352495985745139837002928058309421561388204397335439211554416 9, h=1), inf=Poin
t(x=0, y=0), name='brainpoolP512r1')
EllipticCurve(a=56698187605326110043627228396178346077120614539475214109386828188763884139993, b=175
77232497321838841075697789794520262950426058923084567046852300633325438902, base=Point(x=63243729749
56233335529224355031297033477817557105472658709538162362714411478 6, y=3821861509375352389312227796 40
30810387585405539772602581557831887485717997975), field=Field(p=76884956397045344220809746629001649092737531784414529 53
87555190630635363590 79, h=1), inf=Point(x=0, y=0), name='brainpoolP256r1')
```

In [10]:

```
from typing import List
# common divisor functions -> for natural positive numbers
# returns list of divisors of number n
divs: Callable[[int], List[int]] = lambda n: [i for i in range(1, n + 1)  if n % i == 0]

# interface: GCD and Bezout coefficients (x, y), so ax+bx=gcd(a, b)
@dataclass
class GcdProps:
    gcd: int # prime modulus
    x: int # order of the base point
    y: int = 1

egcdStep: Callable[[int, int, GcdProps], GcdProps] = lambda a, b, g: \
    GcdProps(g.gcd, g.y-(b//a)*g.x, g.x)

# extended GCD euclidean algorithm
egcd: Callable[[int, int], GcdProps] = lambda a, b: \
    GcdProps(b, 0, 1) if a == 0 else \
        egcdStep(a, b, egcd(b%a, a))

# modular multiplicative inverse integer (a/b vs b/a); it is necessary to compute the slope f.ex.
# gcd(a, prime) = x*a + y*prime -> x is the inverse
multInvert: Callable[[EllipticCurve, int], int] = lambda ec, i: \
    0 if i==0 else \
        (ec.field.p - multInvert(ec, -i)) if i < 0 else \
            (lambda g=egcd(i, ec.field.p): None if g.gcd != 1 else g.x%ec.field.p)()

print(egcd(25, 100))
print(egcd(56, 63))
print(egcd(1308, 729))
#print(100000, multInvert(ec1, 100000))
```

```
GcdProps(gcd=25, x=1, y=0)
GcdProps(gcd=7, x=-1, y=1)
GcdProps(gcd=3, x=34, y=-61)
```

In [11]:

```
# check if point is on curve [in the field Fp]
# y^2 % p = (x^3 + Ax + B) % p
isOnCurve: Callable[[EllipticCurve, Point], bool] = lambda ec, p: \
    (p.y**2 - p.x**3 - ec.a*p.x - ec.b) % ec.field.p == 0

# point of infinnity check
isZero: Callable[[EllipticCurve, Point], bool] = lambda ec, p: (p.y == ec.inf.y and p.x == ec.inf.x)

# invert the point (y axis)
invert: Callable[[EllipticCurve, Point], Point] = lambda ec, p: \
    p if isZero(ec, p) else Point(p.x, (-p.y)%ec.field.p)

# check if two points are inverse
areInverse: Callable[[EllipticCurve, Point, Point], bool] = lambda ec, p1, p2: \
    (p1 == invert(ec, p2))
```

In [12]:

```
ec1 = getCurve(ELLIPTIC_CURVES.bp256.value)
G = ec1.base
Gi = invert(ec1, G)

print(areInverse(ec1, G, Gi))
print(isOnCurve(ec1, G))
```

```
True
True
```

In [13]:

```
# slope of the line crossing p1, p2 and inverse of result
# calculates appropriate formula w.r.t if points are on same coordinates
slope: Callable[[EllipticCurve, Point, Point], int] = lambda ec, p1, p2: \
    ((p2.y-p1.y) * multInvert(ec, (p2.x-p1.x)) ) % ec.field.p if (p1 != p2) else \
        ec.inf if p1.y == 0 else \
             ((3*p1.x**2 + ec.a) * multInvert(ec, 2*p1.y)) % ec.field.p

sumPoints: Callable[[EllipticCurve, Point, Point], Point] = lambda ec, p1, p2: \
    (lambda s=slope(ec, p1, p2): \
        (lambda xr=((s**2 - p1.x - p2.x) % ec.field.p): \
            Point(xr, (s * (p1.x-xr) - p1.y) % ec.field.p) \
                )())()

pointAddition: Callable[[EllipticCurve, Point, Point], Point] = lambda ec, p1, p2: \
    p1 if isZero(ec, p2) else p2 if isZero(ec, p1) else \
         ec.inf if areInverse(ec, p1, p2) else \
             sumPoints(ec, p1, p2)

# point doubling
pointDouble: Callable[[EllipticCurve, Point], Point] = lambda ec, p1: \
    pointAddition(ec, p1, p1)

print(pointDouble(ec1, G))
print(pointAddition(ec1, G, G))
```

```
Point(x=52575969560191351534542091466380106041028581718640875237441073011616025668110, y=24843789797109572893402439557748964186754677981311543350228155441542769376468)
Point(x=52575969560191351534542091466380106041028581718640875237441073011616025668110, y=24843789797109572893402439557748964186754677981311543350228155441542769376468)
```

4.4 Double and add algorithm

The ***double-and-add*** (DAA) algoritm computes fast multiplication of the point on the curve with a scalar. In the following steps, the cumulative multiplication as well as DAA will be implemented in order to compare their elapse time.

Furthermore, the ***Montgomery ladder*** approach to DAA computes the point multiplication in a fixed amount of time due to its property of performing the same amount of additions and doubles for any key of the same length[9]. This can prevent leveraging time or power-consumption measurements and their analysis in order to carry out a side-channel attack.

However, it was proven that through application of a *FLUSH+RELOAD* side-channel attack on OpenSSL - one of the most commonly used open-source cryptographic libraries which uses Montgomery ladder in ECDSA - the full private key can be

revealed at a very low cost[10].

4.4.1 Point multiplication with scalar implementations

A function to measure elapse time of function execution was prepared:

In [14]:
```
import time

# returns result and execution time of the passed function as an array
elapseTimeWithResult: Callable[[type(lambda x: None), ...], float] = lambda f, *args: \
    (lambda s=time.process_time(): [f(*args), time.process_time()-s])()

# returns execution time of the passed function
elapseTime: Callable[[type(lambda x: None), ...], float] = lambda f, *args: \
    elapseTimeWithResult(f, *args)[1]
```

At first standard multiplication - following scheme $nP = P + P + \ldots + P\ (n - times)$ was implemented - in order to compare it with the double and add algorithm at later point.

In [15]:
```
# normal multiplication n*P
def pointMultiplySlow(ec: EllipticCurve, p: Point, n: int):
    sum: Point = p
    for x in range(1, n):
        sum = pointAddition(ec, sum, p)
    return sum
```

A function to transform a given number into string of bits (MSB to LSB) and the DAA algorithm. The conversion of the number does not happen inside of the function in order to properly measure execution time of the algorithm itself.

TODO: make a horribly ugly lambda out of it.

In [16]:
```
toBinary: Callable[[int], str] = lambda i: bin(i)[2:]

# standard double and add algorithm
def doubleAndAdd(ec: EllipticCurve, p: Point, bits: str) -> Point:
    sum = ec.inf # starting point: neutral element
    # from MSB to LSB
    for bit in bits:
        sum = pointDouble(ec, sum) # double
        if bit == '1':
            sum = pointAddition(ec, sum, p) # add
    return sum

# standard double and add
#doubleAndAdd: Callable[[EllipticCurve, Point, int], Point] = lambda ec, p, n: \
#    (lambda sum=ec.inf, bits=toBinary(n): [(lambda bit=bit, partSum=pointDouble(ec, sum): \
#        pointAddition(ec1, partSum, p) if bit == '1' else partSum )() for bit in bits])()
```

In [17]:
```
multSlow = pointMultiplySlow(ec1, G, 11)
multFast = doubleAndAdd(ec1, G, toBinary(11))

#print(multSlow == multFast)

reps = 89363

timeMultSlow = elapseTime(pointMultiplySlow, ec1, G, reps)
print('Slow multiplication: ', timeMultSlow, 's')

timeMultFast = elapseTime(doubleAndAdd, ec1, G, toBinary(reps))
print('Fast multiplication (double-add): ', timeMultFast, 's')

print('Difference (slow - fast): ', timeMultSlow - timeMultFast, 's')
```

```
Slow multiplication:  17.094601722 s
Fast multiplication (double-add):  0.004490628000002772 s
Difference (slow - fast):  17.090111093999997 s
```

In [18]:
```
# Montgomery ladder double and add algorithm
def montgomeryDoubleAndAdd(ec: EllipticCurve, p: Point, bits: str) -> Point:
    r0 = ec.inf # sum register
```

```
    r1 = p # intermediate register
    # from MSB to LSB
    for bit in bits:
        if bit == '1':
            r0 = pointAddition(ec, r0, r1)
            r1 = pointDouble(ec, r1)
        else:
            r1 = pointAddition(ec, r0, r1)
            r0 = pointDouble(ec, r0)
    return r0


daaRes1 = doubleAndAdd(ec1, G, toBinary(11))
mlRes1 = montgomeryDoubleAndAdd(ec1, G, toBinary(11))
#print(daaRes1)
#print(mlRes1)
print(daaRes1 == mlRes1)
```

```
True
```

In [35]:

```
# short hand notation for base point of the curve

# slow multiply
slowMultiplyBase: Callable[[EllipticCurve, int], Point] = lambda ec, n: pointMultiplySlow(ec, ec.bas

# double and add
doubleAndAddBase: Callable[[EllipticCurve, str], Point] = lambda ec, bits: doubleAndAdd(ec, ec.base,

# montgomery ladder is the default algorithm for multiplication
pointMultiplication: Callable[[EllipticCurve, str], Point] = lambda ec, bits: montgomeryDoubleAndAdd

# montgomery ladder is the default algorithm for multiplication
anyPointsMultiplication: Callable[[EllipticCurve, Point, str], Point] = lambda ec, p, bits: montgome

# point multiplication (montgomery ladder) with conversion to binary included
pointMultiplyWithInt: Callable[[EllipticCurve, int], Point] = lambda ec, n: montgomeryDoubleAndAdd(e
```

4.4.2 Performance measurements

Functions to generate measurements of the performance of both - standard and Montgomery Ladder - algorithms with regard to the timing of their execution time.

The tests were performed on two Brainpool curves (`brainpoolP256r1` and `brainpoolP512r1`) for 3 datasets, each being a list containing 30 prime numbers in different value ranges, so each set would result in elements with fixed - but diverse between each others - number of bits.

In [36]:

```
import pandas as pd
import numpy as np

# class for calculating common statistics results from any list
@dataclass
class MeasurementsProps:
    data: list
    name: str = 'undefined'
    standardDeviation: float = field(init=False)
    mean: float = field(init=False)
    variance: float = field(init=False)
    median: float = field(init=False)
    max: float = field(init=False)
    min: float = field(init=False)
    def __post_init__(self):
        self.standardDeviation = np.std(self.data)
        self.mean = np.mean(self.data)
        self.variance = np.var(self.data)
        self.median = np.median(self.data)
        self.max = np.amax(self.data)
        self.min = np.amin(self.data)
    def __str__(self):
        return f'- {self.name} {("".join(["-" for _ in range(80-len(self.name))]))} \n\tstandard dev
```

In [37]:

```
def measureDoubleAndAdd(ec: EllipticCurve, scalars: List):
    dataTitleMult = 'Multiplicant'
```

```
    dataTitleMultBin = 'Multiplicant (bin)'
    dataTitleMlTime = 'Montgomery Ladder time'
    dataTitleDaaTime = 'Standard DAA time'
    dataTitleResultEq = 'Results equal'
    # array for data collection
    data = {
        dataTitleMult: [],
        dataTitleMultBin: [],
        dataTitleMlTime: [],
        dataTitleDaaTime: [],
        dataTitleResultEq: []
    }

    for r in scalars:
        rBin = toBinary(r)
        daa = elapseTimeWithResult(doubleAndAddBase, ec, rBin)
        ml = elapseTimeWithResult(pointMultiplication, ec, rBin)
        # save data
        data[dataTitleMult].append(r)
        data[dataTitleMultBin].append(rBin)
        data[dataTitleMlTime].append(ml[1])
        data[dataTitleDaaTime].append(daa[1])
        data[dataTitleResultEq].append(daa[0] == ml[0])

    # pandas data frame for pretty print table
    pd.set_option('display.width', 140)
    df = pd.DataFrame(data)
    # results computation
    mlMeasurementsResults = MeasurementsProps(
        name = 'Montgomery Ladder Timing',
        data = data[dataTitleMlTime]
    )
    daaMeasurementsResults = MeasurementsProps(
        name = 'Standard Double-and-add Timing',
        data = data[dataTitleDaaTime]
    )

    dashes = ''.join( ["=" for _ in range(40-len(ec.name)//2)] )
    print(dashes, 'Curve: ', ec.name, ', samples bit size: ', len(toBinary(scalars[0])), dashes)
    print(mlMeasurementsResults)
    print(daaMeasurementsResults)
    print(df)
    print()


# data samples: 30 primes from different bit-length ranges
primes1 = [
    11731, 11743, 11777, 11779, 11783, 11789, 11801, 11807, 11813, 11821,
    11827, 11831, 11833, 11839, 11863, 11867, 11887, 11897, 11903, 11909,
    11923, 11927, 11933, 11939, 11941, 11953, 11959, 11969, 11971, 11981
]

primes2 = [
    284059, 284083, 284093, 284111, 284117, 284129, 284131, 284149, 284153, 284159,
    284161, 284173, 284191, 284201, 284227, 284231, 284233, 284237, 284243, 284261,
    284267, 284269, 284293, 284311, 284341, 284357, 284369, 284377, 284387, 284407
]

primes3 = [
    14837491, 14837503, 14837513, 14837519, 14837527, 14837533, 14837539, 14837551, 14837587, 148375
    14837593, 14837609, 14837653, 14837657, 14837663, 14837677, 14837681, 14837687, 14837707, 148377
    14837743, 14837789, 14837807, 14837839, 14837857, 14837861, 14837873, 14837891, 14837897, 148378
]


ecBp256 = getCurve(ELLIPTIC_CURVES.bp256.value)
ecBp512 = getCurve(ELLIPTIC_CURVES.bp512.value)

measureDoubleAndAdd(ecBp256, primes1)
measureDoubleAndAdd(ecBp512, primes1)
measureDoubleAndAdd(ecBp256, primes2)
measureDoubleAndAdd(ecBp512, primes2)
measureDoubleAndAdd(ecBp256, primes3)
measureDoubleAndAdd(ecBp512, primes3)
```

```
================================ Curve:  brainpoolP256r1 , samples bit size:  14 ================
===============
```

```
- Montgomery Ladder Timing ------------------------------------------------------
        standard deviation: 0.0007860222000046378
        variance: 6.178308989001308e-07
        arithmetic mean: 0.00510775866666601
        median: 0.004651086000006188
        maximum: 0.007175843999995379
        minimum: 0.004564157000004343

- Standard Double-and-add Timing ------------------------------------------------
        standard deviation: 0.0006034981624561635
        variance: 3.642100320879659e-07
        arithmetic mean: 0.0037977051333342613
        median: 0.003578678499998489
        maximum: 0.0062158460000034665
        minimum: 0.0031908280000010336

    Multiplicant Multiplicant (bin)  Montgomery Ladder time  Standard DAA time  Results equal
0          11731     10110111010011                0.006233           0.005081           True
1          11743     10110111011111                0.007176           0.006216           True
2          11777     10111000000001                0.006902           0.004003           True
3          11779     10111000000011                0.006416           0.004246           True
4          11783     10111000000111                0.005984           0.004175           True
5          11789     10111000001101                0.006105           0.004134           True
6          11801     10111000011001                0.005008           0.003713           True
7          11807     10111000011111                0.004886           0.003790           True
8          11813     10111000100101                0.004848           0.003424           True
9          11821     10111000101101                0.004755           0.003562           True
10         11827     10111000110011                0.004649           0.003382           True
11         11831     10111000110111                0.004602           0.003524           True
12         11833     10111000111001                0.004625           0.003309           True
13         11839     10111000111111                0.004784           0.003683           True
14         11863     10111001010111                0.004995           0.003586           True
15         11867     10111001011011                0.004957           0.003899           True
16         11887     10111001101111                0.004595           0.003960           True
17         11897     10111001111001                0.004569           0.003571           True
18         11903     10111001111111                0.004568           0.003842           True
19         11909     10111010000101                0.004626           0.003255           True
20         11923     10111010010011                0.004607           0.003384           True
21         11927     10111010010111                0.004635           0.003569           True
22         11933     10111010011101                0.004603           0.003545           True
23         11939     10111010100011                0.004581           0.003350           True
24         11941     10111010100101                0.004564           0.003321           True
25         11953     10111010110001                0.006477           0.004507           True
26         11959     10111010110111                0.004653           0.003762           True
27         11969     10111011000001                0.004621           0.003191           True
28         11971     10111011000011                0.004578           0.003416           True
29         11981     10111011001101                0.004628           0.003531           True

================================ Curve:  brainpoolP512r1 , samples bit size:  14 ==================
===============
- Montgomery Ladder Timing ------------------------------------------------------
        standard deviation: 0.0008966889252673449
        variance: 8.040510286971061e-07
        arithmetic mean: 0.011110553500000009
        median: 0.010847151499998375
        maximum: 0.014085552999993922
        minimum: 0.009956111000001044

- Standard Double-and-add Timing ------------------------------------------------
        standard deviation: 0.0010359271446680844
        variance: 1.0731450490601702e-06
        arithmetic mean: 0.008396928300000184
        median: 0.008230667999999497
        maximum: 0.010651060000000712
        minimum: 0.006423644000001616

    Multiplicant Multiplicant (bin)  Montgomery Ladder time  Standard DAA time  Results equal
0          11731     10110111010011                0.010265           0.008684           True
1          11743     10110111011111                0.010549           0.009734           True
2          11777     10111000000001                0.009956           0.006424           True
3          11779     10111000000011                0.010500           0.006647           True
4          11783     10111000000111                0.010958           0.007349           True
5          11789     10111000001101                0.010453           0.007410           True
6          11801     10111000011001                0.011157           0.007510           True
7          11807     10111000011111                0.010621           0.008290           True
8          11813     10111000100101                0.010847           0.007318           True
9          11821     10111000101101                0.011088           0.008171           True
```

```
10         11827     10111000110011               0.010847           0.007890           True
11         11831     10111000110111               0.011205           0.008648           True
12         11833     10111000111001               0.010789           0.008402           True
13         11839     10111000111111               0.011174           0.010095           True
14         11863     10111001010111               0.011705           0.008427           True
15         11867     10111001011011               0.012407           0.009122           True
16         11887     10111001101111               0.011783           0.010035           True
17         11897     10111001111001               0.010577           0.008745           True
18         11903     10111001111111               0.010800           0.008854           True
19         11909     10111010000101               0.010666           0.007541           True
20         11923     10111010010011               0.010281           0.007662           True
21         11927     10111010010111               0.010342           0.007977           True
22         11933     10111010011101               0.010127           0.007896           True
23         11939     10111010100011               0.010252           0.007570           True
24         11941     10111010100101               0.011448           0.008023           True
25         11953     10111010110001               0.011616           0.008007           True
26         11959     10111010110111               0.012208           0.009508           True
27         11969     10111011000001               0.013050           0.009227           True
28         11971     10111011000011               0.014086           0.010089           True
29         11981     10111011001101               0.011561           0.010651           True

================================ Curve:  brainpoolP256r1 , samples bit size:  19 ==================
===============
- Montgomery Ladder Timing ------------------------------------------------------
        standard deviation: 0.000314967117635537
        variance: 9.920428519163818e-08
        arithmetic mean: 0.006785067099999746
        median: 0.006734626999996607
        maximum: 0.007599298999998894
        minimum: 0.0063335160000050905

- Standard Double-and-add Timing ------------------------------------------------
        standard deviation: 0.000440120717298964
        variance: 1.9370624579575457e-07
        arithmetic mean: 0.0050208886666657316
        median: 0.005015507999999613
        maximum: 0.005861729999999454
        minimum: 0.004235522999998409

    Multiplicant   Multiplicant (bin)  Montgomery Ladder time  Standard DAA time  Results equal
0         284059  1000101010110011011                0.007599           0.005599           True
1         284083  1000101010110110011                0.006894           0.005726           True
2         284093  1000101010110111101                0.007087           0.005119           True
3         284111  1000101010111001111                0.006428           0.004890           True
4         284117  1000101010111010101                0.006334           0.004990           True
5         284129  1000101010111100001                0.006825           0.004654           True
6         284131  1000101010111100011                0.006497           0.004711           True
7         284149  1000101010111110101                0.006370           0.005196           True
8         284153  1000101010111111001                0.006650           0.005162           True
9         284159  1000101010111111111                0.006849           0.005244           True
10        284161  1000101011000000001                0.006474           0.004236           True
11        284173  1000101011000001101                0.006605           0.004713           True
12        284191  1000101011000011111                0.006792           0.004774           True
13        284201  1000101011000101001                0.006520           0.004561           True
14        284227  1000101011001000011                0.006592           0.004694           True
15        284231  1000101011001000111                0.006652           0.004566           True
16        284233  1000101011001001001                0.006369           0.004482           True
17        284237  1000101011001001101                0.006603           0.004730           True
18        284243  1000101011001010011                0.007096           0.005041           True
19        284261  1000101011001100101                0.006943           0.004726           True
20        284267  1000101011001101011                0.006530           0.005485           True
21        284269  1000101011001101101                0.006776           0.005754           True
22        284293  1000101011010000101                0.006694           0.004360           True
23        284311  1000101011010010111                0.007455           0.005043           True
24        284341  1000101011010110101                0.006830           0.005494           True
25        284357  1000101011011000101                0.007067           0.005072           True
26        284369  1000101011011010001                0.006970           0.004728           True
27        284377  1000101011011011001                0.006665           0.005210           True
28        284387  1000101011011100011                0.007068           0.005862           True
29        284407  1000101011011110111                0.007319           0.005806           True

================================ Curve:  brainpoolP512r1 , samples bit size:  19 ==================
===============
- Montgomery Ladder Timing ------------------------------------------------------
        standard deviation: 0.0015347327211566943
        variance: 2.355404525389032e-06
        arithmetic mean: 0.014843036999998798
```

```
        median: 0.014206681500002816
        maximum: 0.019360505999998168
        minimum: 0.013165112999999451

- Standard Double-and-add Timing -------------------------------------------------
        standard deviation: 0.0014658536476877435
        variance: 2.1487269164394632e-06
        arithmetic mean: 0.01082193733333246
        median: 0.010357449500006055
        maximum: 0.016420027999998865
        minimum: 0.008657729999995922
```

	Multiplicant	Multiplicant (bin)	Montgomery Ladder time	Standard DAA time	Results equal
0	284059	1000101010110011011	0.017748	0.012831	True
1	284083	1000101010110110011	0.019361	0.012606	True
2	284093	1000101010110111101	0.017115	0.012993	True
3	284111	1000101010111001111	0.018508	0.016420	True
4	284117	1000101010111010101	0.014397	0.010718	True
5	284129	1000101010111100001	0.013920	0.010167	True
6	284131	1000101010111100011	0.014104	0.010099	True
7	284149	1000101010111110101	0.014348	0.011076	True
8	284153	1000101010111111001	0.013973	0.010522	True
9	284159	1000101010111111111	0.014072	0.012231	True
10	284161	1000101011000000001	0.013859	0.008658	True
11	284173	1000101011000001101	0.013920	0.009972	True
12	284191	1000101011000011111	0.013881	0.010517	True
13	284201	1000101011000101001	0.014418	0.010284	True
14	284227	1000101011001000011	0.014305	0.009302	True
15	284231	1000101011001000111	0.015618	0.009902	True
16	284233	1000101011001001001	0.016012	0.010260	True
17	284237	1000101011001001101	0.016873	0.012072	True
18	284243	1000101011001010011	0.015456	0.011959	True
19	284261	1000101011001100101	0.014519	0.010260	True
20	284267	1000101011001101011	0.014364	0.010434	True
21	284269	1000101011001101101	0.014332	0.010235	True
22	284293	1000101011010000101	0.013670	0.009653	True
23	284311	1000101011010010111	0.013850	0.010431	True
24	284341	1000101011010110101	0.013641	0.010071	True
25	284357	1000101011011000101	0.014052	0.009990	True
26	284369	1000101011011010001	0.013165	0.009807	True
27	284377	1000101011011011001	0.013834	0.010503	True
28	284387	1000101011011100011	0.013869	0.009715	True
29	284407	1000101011011110111	0.014108	0.010971	True

```
================================ Curve:  brainpoolP256r1 , samples bit size:  24 ==================
===============
- Montgomery Ladder Timing -------------------------------------------------------
        standard deviation: 0.00084639611939219
        variance: 7.163863909221582e-07
        arithmetic mean: 0.009032804700000223
        median: 0.008743536999997303
        maximum: 0.011602218000000164
        minimum: 0.008167284000000244

- Standard Double-and-add Timing -------------------------------------------------
        standard deviation: 0.0009148482496189037
        variance: 8.369473198307719e-07
        arithmetic mean: 0.006736056166666534
        median: 0.006380538500000199
        maximum: 0.00887454799999432
        minimum: 0.005548488000002294
```

	Multiplicant	Multiplicant (bin)	Montgomery Ladder time	Standard DAA time	Results equal
0	14837491	11100010011001011110011	0.008955	0.007599	True
1	14837503	11100010011001011111111	0.009180	0.008657	True
2	14837513	11100010011001100001001	0.009699	0.008875	True
3	14837519	11100010011001100001111	0.010875	0.008696	True
4	14837527	11100010011001100010111	0.011602	0.008000	True
5	14837533	11100010011001100011101	0.010298	0.008125	True
6	14837539	11100010011001100100011	0.009153	0.006853	True
7	14837551	11100010011001100101111	0.008783	0.006634	True
8	14837587	11100010011001101010011	0.008364	0.006155	True
9	14837591	11100010011001101010111	0.008167	0.006218	True
10	14837593	11100010011001101011001	0.008413	0.006358	True
11	14837609	11100010011001101101001	0.008285	0.006178	True
12	14837653	11100010011001110010101	0.008622	0.006123	True
13	14837657	11100010011001110011001	0.008891	0.006329	True

```
14      14837663  11100010011001111110011111               0.008557           0.006501           True
15      14837677  11100010011001111110101101               0.008788           0.006348           True
16      14837681  11100010011001111110110001               0.008365           0.006086           True
17      14837687  11100010011001111110110111               0.009899           0.006403           True
18      14837707  11100010011001111111001011               0.008416           0.007052           True
19      14837723  11100010011001111111011011               0.008345           0.006562           True
20      14837743  11100010011001111111101111               0.008287           0.007424           True
21      14837789  11100010011010000001101101               0.008330           0.005974           True
22      14837807  11100010011010000001011111               0.008315           0.006007           True
23      14837839  11100010011010000101001111               0.008705           0.005775           True
24      14837857  11100010011010000110000101               0.008386           0.005548           True
25      14837861  11100010011010000110010101               0.008423           0.005587           True
26      14837873  11100010011010000111000101               0.009276           0.005926           True
27      14837891  11100010011010001000001101               0.010014           0.006284           True
28      14837897  11100010011010001000100101               0.009673           0.006705           True
29      14837899  11100010011010001000101011               0.009919           0.007100           True

================================= Curve:  brainpoolP512r1 , samples bit size:  24 ==================
===============
- Montgomery Ladder Timing ------------------------------------------------------------
        standard deviation: 0.0027253905156750594
        variance: 7.427753462931566e-06
        arithmetic mean: 0.019334427799999315
        median: 0.018336880999996197
        maximum: 0.03005349999999396
        minimum: 0.017872714999995765

- Standard Double-and-add Timing ------------------------------------------------
        standard deviation: 0.0018244703527386312
        variance: 3.3286920680222253e-06
        arithmetic mean: 0.014076661933332938
        median: 0.013845650500002193
        maximum: 0.02169620600000144
        minimum: 0.012234827999996867

    Multiplicant        Multiplicant (bin)  Montgomery Ladder time  Standard DAA time  Results equal
0       14837491  11100010011001011110011               0.020438           0.018391           True
1       14837503  11100010011001011111111               0.018371           0.015647           True
2       14837513  11100010011001100001001               0.018250           0.013165           True
3       14837519  11100010011001100001111               0.018264           0.013903           True
4       14837527  11100010011001100010111               0.018322           0.013869           True
5       14837533  11100010011001100011101               0.018295           0.013823           True
6       14837539  11100010011001100100011               0.018826           0.013965           True
7       14837551  11100010011001100101111               0.018096           0.013789           True
8       14837587  11100010011001101010011               0.021999           0.014157           True
9       14837591  11100010011001101010111               0.019804           0.014383           True
10      14837593  11100010011001101011001               0.027936           0.021696           True
11      14837609  11100010011001101101001               0.019070           0.014330           True
12      14837653  11100010011001110010101               0.018418           0.013691           True
13      14837657  11100010011001110011001               0.018256           0.013661           True
14      14837663  11100010011001110011111               0.018300           0.014073           True
15      14837677  11100010011001110101101               0.017873           0.013546           True
16      14837681  11100010011001110110001               0.017882           0.013222           True
17      14837687  11100010011001110110111               0.017940           0.013931           True
18      14837707  11100010011001111001011               0.018072           0.013575           True
19      14837723  11100010011001111011011               0.018310           0.014256           True
20      14837743  11100010011001111101111               0.018674           0.014695           True
21      14837789  11100010011010000001101               0.019338           0.012803           True
22      14837807  11100010011010000001011111               0.030053           0.014758           True
23      14837839  11100010011010000101001111               0.019105           0.013989           True
24      14837857  11100010011010000110000101               0.018214           0.012290           True
25      14837861  11100010011010000110010101               0.018338           0.012569           True
26      14837873  11100010011010000111000101               0.018564           0.012435           True
27      14837891  11100010011010001000001101               0.018336           0.012235           True
28      14837897  11100010011010001000100101               0.018320           0.012420           True
29      14837899  11100010011010001000101011               0.018371           0.013034           True
```

The timing analysis showed that the Montgomery ladder was mostly slower than the standard Double-and-add algorithm, which was expected. As well the standard deviation was lower in case of the Montgomery ladder version, pointing to more time-invariant results.

Both algorithms always yielded the same result of the multiplication.

4.5 ECDH: Implementation of key manager model

A simple key management model was implemented as a dataclass which holds an id of the user, the owner's public-private key pair and a list of shared keys (object consisting of an id of another user and calculated with)

In [38]:
```
@dataclass
class KeyPair:
    public: Point
    private: int

@dataclass
class SharedKey:
    key: Point
    sharedWith: str
    def __str__(self):
        return f'shared with: {self.sharedWith}, key: {self.key}\n'

@dataclass
class KeyManager:
    ownKeys: KeyPair
    shared: list # list of shared keys
    size: int
    id: str = ''
    def __str__(self):
        return f'--- KEY MANAGER ---\nowner: {self.id} \nkeys size: {self.size} \npublic key: {self.
```

In [39]:
```
# ECDH helper functions algorithm
import secrets

#def generateKeyPair(curve: EllipticCurve) -> KeyPair:
#    priv = secrets.randbelow(curve.field.p) # large random number below prime modulus of the field
#    pub = pointMultiplication(curve, toBinary(priv))
#    return KeyPair(public=pub, private=priv)


#def getCurveForKeySize(size: int):
#    if (size == 256):
#        curve = getCurve(ELLIPTIC_CURVES.bp256.value)
#    elif (size == 512):
#        curve = getCurve(ELLIPTIC_CURVES.bp512.value)
#    else:
#        raise ValueError('Invalid key size; pick 256 or 512.')

generatePrivateKey: Callable[[EllipticCurve], int] = lambda ec: secrets.randbelow(ec.field.p)

generatePublicKeyFromPrivate: Callable[[EllipticCurve, Point], int] = lambda ec, priv: pointMultipli

generateKeyPair: Callable[[EllipticCurve], KeyPair] = lambda ec: (lambda priv=generatePrivateKey(ec)
    KeyPair(public=pointMultiplication(ec, toBinary(priv)), private=priv))()

getCurveForKeySize: Callable[[int], EllipticCurve] = lambda size: \
    getCurve(ELLIPTIC_CURVES.bp256.value) if size == 256 else \
        getCurve(ELLIPTIC_CURVES.bp512.value) if size == 512 else None

getSharedKey: Callable[[EllipticCurve, int, Point, id], SharedKey] = lambda ec, a_priv, b_pub, b_id:
    SharedKey(anyPointsMultiplication(ec, b_pub, toBinary(a_priv)), b_id)
```

In [40]:
```
@dataclass
class EcdhReturn:
    a: KeyManager
    b: KeyManager
# function to generate two key pairs and calculate the shared key between them
# the shared key is calculated twice in order to verify that the results are equal.
def ecdhWithKeyGen(size: int, aId: str, bId: str) -> EcdhReturn:
    curve = getCurveForKeySize(size)
    if (curve is None): raise ValueError('Invalid key size; pick 256 or 512.')
    keysA = generateKeyPair(curve)
    keysB = generateKeyPair(curve)

    sharedForA = SharedKey(anyPointsMultiplication(curve, keysB.public, toBinary(keysA.private)), bI
    sharedForB = SharedKey(anyPointsMultiplication(curve, keysA.public, toBinary(keysB.private)), aI

    return EcdhReturn(
        a=KeyManager(id=aId, size=size, ownKeys=keysA, shared=[sharedForA]),
        b=KeyManager(id=bId, size=size, ownKeys=keysB, shared=[sharedForB])
```

```
    )

# check

a_id = 'a-key-1'
b_id = 'b-key-1'

keys = ecdhWithKeyGen(256, a_id, b_id)

print(keys.a)
print(keys.b)
```

```
--- KEY MANAGER ---
owner: a-key-1
keys size: 256
public key: Point(x=30147845480387316664590173668972599713641593001161892243164578955697811163956, y
=185790933365576279483501204807839100236707289427582853031453699114382742983O4)
private: 533607362778870113724130800478718989992525475352044162648426190426610550317 48
shared keys:
  # shared with: b-key-1, key: Point(x=2363092856192500083087414066610041142546825771410035077612332
7485697861860049, y=2939082303376902800487265678034822836324243828219085366949879717102629996117O)

--- KEY MANAGER ---
owner: b-key-1
keys size: 256
public key: Point(x=70517931307624186814125767378483331378282109296193646495459520518596538595182, y
=154690115523797375565832589672697062974826506921870773458720263424943445538 55)
private: 49757647874544693904937988996843043236537964401698010887024841177116351918467
shared keys:
  # shared with: a-key-1, key: Point(x=2363092856192500083087414066610041142546825771410035077612332
7485697861860049, y=2939082303376902800487265678034822836324243828219085366949879717102629996117O)
```

ECDH: Brainpool Test Vectors

The data of oficially published [11] test vectors of Brainpool curves was collected for keys with length of 256 and 512 bit. Followingly, the functions to compute the public key from the private one, as well as the shared key calculation were tested.

In [44]:

```
@dataclass
class BrainpoolTestVector():
    a: KeyPair
    b: KeyPair
    sh: SharedKey

def getBrainpoolTestVector(a_priv: int, a_x: int, a_y: int, b_priv: int, b_x: int, b_y: int, sh_x: i
    return BrainpoolTestVector(
        a=KeyPair(
            public=Point(
                x=a_x,
                y=a_y
            ),
            private=a_priv
        ),
        b=KeyPair(
            public=Point(
                x=b_x,
                y=b_y
            ),
            private=b_priv
        ),
        sh=SharedKey(
            key=Point(
                x=sh_x,
                y=sh_y
            ),
            sharedWith='test-vector'
        ),
    )

bp256_testVector = getBrainpoolTestVector(
    a_x=0x78028496B5ECAAB3C8B6C12E45DB1E02C9E4D26B4113BC4F015F60C5CCC0D206,
    a_y=0xA2AE1762A3831C1D20F03F8D1E3C0C39AFE6F09B4D44BBE80CD100987B05F92B,
    a_priv=0x041EB8B1E2BC681BCE8E39963B2E9FC415B05283313DD1A8BCC055F11AE49699,
    b_x=0x8E07E219BA588916C5B06AA30A2F464C2F2ACFC1610A3BE2FB240B635341F0DB,
    b_y=0x148EA1D7D1E7E54B9555B6C9AC90629C18B63BEE5D7AA6949EBBF47B24FDE40D,
```

```
    b_priv=0x06F5240EACDB9837BC96D48274C8AA834B6C87BA9CC3EEDD81F99A16B8D804D3,
    sh_x=0x05E940915549E9F6A4A75693716E37466ABA79B4BF2919877A16DD2CC2E23708,
    sh_y=0x6BC23B6702BC5A019438CEEA107DAAD8B94232FFBBC350F3B137628FE6FD134C
)

bp512_testVector = getBrainpoolTestVector(
    a_x=0x0A420517E406AAC0ACDCE90FCD71487718D3B953EFD7FBEC5F7F27E28C6149999397E91E029E06457DB2D3E640
    a_y=0x72E6882E8DB28AAD36237CD25D580DB23783961C8DC52DFA2EC138AD472A0FCEF3887CF62B623B2A87DE5C5883
    a_priv=0x16302FF0DBBB5A8D733DAB7141C1B45ACBC8715939677F6A56850A38BD87BD59B09E80279609FF333EB9D4C
    b_x=0x9D45F66DE5D67E2E6DB6E93A59CE0BB48106097FF78A081DE781CDB31FCE8CCBAAEA8DD4320C4119F1E9CD437A
    b_y=0x2FDC313095BCDD5FB3A91636F07A959C8E86B5636A1E930E8396049CB481961D365CC11453A06C719835475B12
    b_priv=0x230E18E1BCC88A362FA54E4EA3902009292F7F8033624FD471B5D8ACE49D12CFABBC19963DAB8E2F1EBA00E
    sh_x=0xA7927098655F1F9976FA50A9D566865DC530331846381C87256BAF3226244B76D36403C024D7BBF0AA0803EAF
    sh_y=0x7DB71C3DEF63212841C463E881BDCF055523BD368240E6C3143BD8DEF8B3B3223B95E0F53082FF5E412F42225
)
```

In [48]:

```
import unittest

class TestEcdh(unittest.TestCase):

    def testPublicKeys(self):
        bp256 = getCurve(ELLIPTIC_CURVES.bp256.value)
        bp512 = getCurve(ELLIPTIC_CURVES.bp512.value)

        keyA_256 = generatePublicKeyFromPrivate(bp256, bp256_testVector.a.private)
        keyA_512 = generatePublicKeyFromPrivate(bp512, bp512_testVector.a.private)

        keyB_256 = generatePublicKeyFromPrivate(bp256, bp256_testVector.b.private)
        keyB_512 = generatePublicKeyFromPrivate(bp512, bp512_testVector.b.private)

        self.assertEqual(keyA_256, bp256_testVector.a.public)
        self.assertEqual(keyA_512, bp512_testVector.a.public)
        self.assertEqual(keyB_256, bp256_testVector.b.public)
        self.assertEqual(keyB_512, bp512_testVector.b.public)


    def testSharedKey(self):
        bp256 = getCurve(ELLIPTIC_CURVES.bp256.value)
        bp512 = getCurve(ELLIPTIC_CURVES.bp512.value)
        sharedKey256 = getSharedKey(bp256, bp256_testVector.a.private, bp256_testVector.b.public, 't
        sharedKey512 = getSharedKey(bp512, bp512_testVector.a.private, bp512_testVector.b.public, 't
        self.assertEqual(sharedKey256.key, bp256_testVector.sh.key)
        self.assertEqual(sharedKey512.key, bp512_testVector.sh.key)


unittest.main(argv=[''], verbosity=5, exit=False)
```

```
testPublicKeys (__main__.TestEcdh) ... ok
testSharedKey (__main__.TestEcdh) ... ok

----------------------------------------------------------------------
Ran 2 tests in 1.503s

OK
```

Out[48]:

```
<unittest.main.TestProgram at 0x7f7937461460>
```

References

[1] Ritzenhaler C., *Introduction to elliptic curves*, Université de Rennes, 2013-2014, online access: https://perso.univ-rennes1.fr/christophe.ritzenthaler/cours/elliptic-curve-course.pdf

[2] Svetlin Nakov, PhD, *Practical cryptography for developers: Elliptic Curve Cryptography (ECC)*, online-access: https://cryptobook.nakov.com/asymmetric-key-ciphers/elliptic-curve-cryptography-ecc

[3] Lochter M., Merkle J., Schmidt J.M., Schuetze T., *Requirements for Elliptic Curves for High-Assurance Applications*, chapter *3.5 Implementation Security*, online-access: https://csrc.nist.gov/csrc/media/events/workshop-on-elliptic-curve-cryptography-standards/documents/papers/session4-merkle-johannes.pdf

[4] Barker E., Chen L., Roginsky A., Smid M., *Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography* NIST Special Publication 800-56A Revision 2, 2013, online-access:

https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Ar2.pdf

[5] Chen L., Moody D., Regenschied A., Randall K., *Recommendations for Discrete Logarithm-Based Cryptography: Elliptic Curve Domain Parameters*, p. 9, 2019, online-access: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-186-draft.pdf

[6] Federal Information Processing Standards Publication, *Digital Signature Standard (DSS)*, FIPS PUB 186-4, *Appendix D: Recommended Elliptic Curves for Federal Government Use*, p. 87, online-access: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf

[7] PEP-557, *Data classes*, online-access: https://www.python.org/dev/peps/pep-0557/

[8] Lochter M., Merkle J., *Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation*, chapter *3 Domain Parameter Specification*, 2010, IETF Trust, online-access: https://www.rfc-editor.org/rfc/pdfrfc/rfc5639.txt.pdf

[9] Gueron Sh., Paar Ch., *Software optimization of binary elliptic curves arithmetic using modern processor architectures*, 2013, Ruhr University Bochum, p. 14, online-access: https://www.emsec.ruhr-uni-bochum.de/media/attachments/files/2014/11/MA_Bluhm.pdf

[10] Yarom Y., Benger N., *Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack*, 2014, online-access: https://eprint.iacr.org/2014/140.pdf

[11] *RFC 8734 Elliptic Curve Cryptography (ECC) Brainpool Curves for Transport Layer Security (TLS) Version 1.3*, Appendix A. Test Vectors, online-access: https://www.rfc-editor.org/rfc/rfc8734.html#section-appendix.a-1 / https://datatracker.ietf.org/doc/html/rfc6932#appendix-A.1

Other resources:

Elliptic Curve Cryptography, Technical Guideline BSI TR-03111, Bundesamt fur Sicherheit in der Informationstechnik, 2018, online-access: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TR03111/BSI-TR-03111_V-2-1_pdf.pdf?__blob=publicationFile&v=1

Buchanan B., *Barebones P256: There Is Beauty in Elliptic Curves*, 2020, online-access: https://billatnapier.medium.com/barebones-p256-1700ff5a4

Dharmapurikar M., Jandhyala A., Jones S., Renaud C., *Elliptic Curve Cryptography*, 2018, online-access: https://girlstalkmath.com/2018/07/02/elliptic-curve-cryptography-2/