

MTAT.07.017
Applied Cryptography

Elliptic Curve Cryptography (ECC)

University of Tartu

Spring 2023

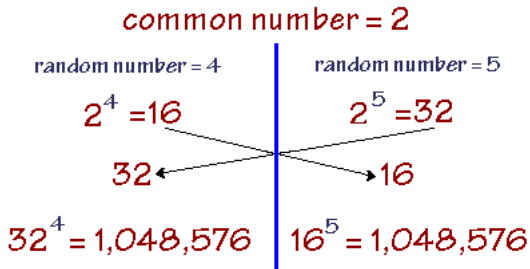
Diffie-Hellman key exchange



Ralph Merkle, Martin Hellman, Whitfield Diffie (1976)

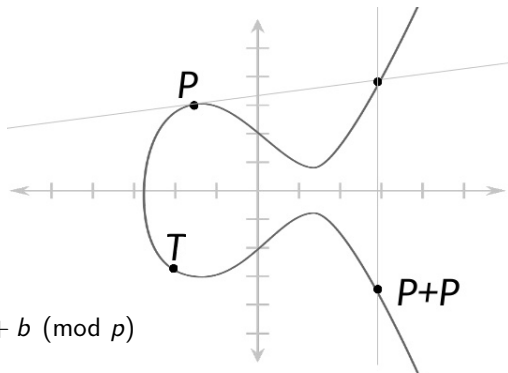
- The first public-key cryptography algorithm

Diffie-Hellman (DH) key exchange



- $(2^5)^4 = 2^{5 \cdot 4} = (2^4)^5$
- In practice: multiplicative group of integers modulo prime p is used
- Discrete logarithm problem:
 - hard to find x , given $2^x = 32 \pmod{p}$
- ElGamal and DSA are based on DL problem
- Secure against passive eavesdropping

Elliptic Curve Cryptography



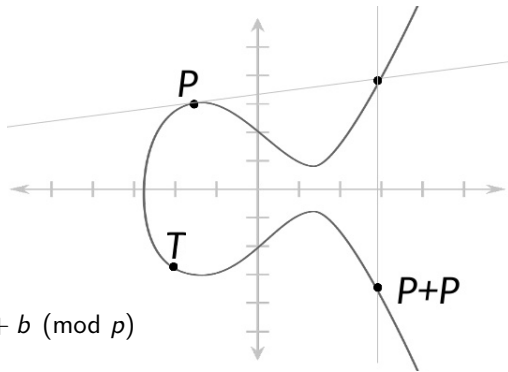
$$y^2 \pmod{p} = x^3 + ax + b \pmod{p}$$

- Formulas for point addition, doubling
- DL problem: find x , given $a^x = b \pmod{p}$
- ECDL problem: find x , given $\underbrace{P + P + \dots + P}_{x \text{ times}} = x \cdot P = T$
- Security: 256-bit ECC $\approx$ 3072-bit RSA $\approx$ 128-bit security level

Standard (named) elliptic curves

- secp256k1
- NIST P-256 / secp256r1 / prime256v1
- NIST P-384 / secp384r1
- NIST P-521 / secp521r1
- brainpoolP256r1
- brainpoolP384r1
- brainpoolP512r1
- Curve25519
- Ed25519

EC domain parameters



$$y^2 \pmod{p} = x^3 + ax + b \pmod{p}$$

- p – field size
- a, b – equation parameters
- G – generator (base point)
- n – order of the curve (total number of points on curve)
- h – cofactor (number of cyclic subgroups)

EC domain parameters

```
$ openssl ecparam -name prime256v1 -text -param_enc explicit -no_seed --noout
```

```
Field Type: prime-field
```

```
Prime:
```

```
00:ff:ff:ff:ff:00:00:00:01:00:00:00:00:00:00:
00:00:00:00:00:00:ff:ff:ff:ff:ff:ff:ff:ff:
ff:ff:ff
```

```
A:
```

```
00:ff:ff:ff:ff:00:00:00:01:00:00:00:00:00:00:
00:00:00:00:00:00:ff:ff:ff:ff:ff:ff:ff:ff:
ff:ff:fc
```

```
B:
```

```
5a:c6:35:d8:aa:3a:93:e7:b3:eb:bd:55:76:98:86:
bc:65:1d:06:b0:cc:53:b0:f6:3b:ce:3c:3e:27:d2:
60:4b
```

```
Generator (uncompressed):
```

```
04:6b:17:d1:f2:e1:2c:42:47:f8:bc:e6:e5:63:a4:
40:f2:77:03:7d:81:2d:eb:33:a0:f4:a1:39:45:d8:
98:c2:96:4f:e3:42:e2:fe:1a:7f:9b:8e:e7:eb:4a:
7c:0f:9e:16:2b:ce:33:57:6b:31:5e:ce:cb:b6:40:
68:37:bf:51:f5
```

```
Order:
```

```
00:ff:ff:ff:ff:00:00:00:00:ff:ff:ff:ff:ff:ff:
ff:ff:bc:e6:fa:ad:a7:17:9e:84:f3:b9:ca:c2:fc:
63:25:51
```

```
Cofactor: 1 (0x1)
```

EC point operations

- Point addition
 - $P + Q = R$

$$\lambda = \frac{y_q - y_p}{x_q - x_p}$$

$$x_r = \lambda^2 - x_p - x_q$$

$$y_r = \lambda(x_p - x_r) - y_p$$

- Point doubling
 - $P + P$

$$\lambda = \frac{3x_p^2 + a}{2y_p}$$

- Point multiplication
 - $k \times P$

“double-and-add”

- Point at infinity (∞)
 - $n \times P = \infty$

- Point negation
 - $P + (-P) = \infty$

$$-P = (x_p, p - y_p)$$

- Point validation

- Check whether the point is on the curve: $y^2 \pmod{p} \stackrel{?}{=} x^3 + ax + b \pmod{p}$

Elliptic Curve Diffie-Hellman (ECDH)

Alice	Bob
Key generation	
$d_a \xleftarrow{\$} [1, n - 1]$	$d_b \xleftarrow{\$} [1, n - 1]$
$Q_a = d_a \times G$	$Q_b = d_b \times G$
	$Q_a \longrightarrow$
	$\longleftarrow Q_b$
Validate Q_b	Validate Q_a
Shared secret calculation	
$Q_{ab} = d_a \times Q_b$	$Q_{ab} = d_b \times Q_a = d_b \times (d_a \times G)$

- Shared secret: x coordinate of Q_{ab}
- ECDH $\rightarrow$ shared secret $\rightarrow$ hybrid encryption

EC private key file format

```
$ openssl genpkey -algorithm EC -pkeyopt ec_paramgen_curve:prime256v1 -out priv.pem
```

```
$ cat priv.pem
```

```
-----BEGIN PRIVATE KEY-----
```

```
MIGHAgEAMBMGBYqGSM49AgEGCCqGSM49AwEHBGOWawIBAQQgrFXchvnjGXn+Xmpo
```

```
7NXSNXgrPNfJRC4Ypx53X7Hn/3TbBJysAGGocPi3BWBL03m0iFOukNAR
```

```
-----END PRIVATE KEY-----
```

```
$ openssl pkcs8 -topk8 -in priv.pem -outform der -out priv.der -nocrypt
```

```
$ dumpasn1 priv.der
```

```
0 135: SEQUENCE {
  3 1:  INTEGER 0
  6 19:  SEQUENCE {
    8 7:  OBJECT IDENTIFIER ecPublicKey (1 2 840 10045 2 1)
  17 8:  OBJECT IDENTIFIER prime256v1 (1 2 840 10045 3 1 7)
    :    }
  27 109: OCTET STRING, encapsulates {
  29 107:  SEQUENCE {
    31 1:  INTEGER 1
    34 32: OCTET STRING
      :    AC 55 DC 86 F9 E3 19 79 FE 5E 6A 68 2B 57 3C 6A
      :    96 AD 31 27 53 B7 65 76 48 20 00 7E 4D 83 7B 74
  68 68:  [1] {
  70 66:  BIT STRING
      :    04 39 E0 CE 3D 25 F8 04 33 FF E6 A9 E4 5C 81 DC
      :    6E 51 A6 00 26 58 CE EC D5 D2 35 78 11 3C D7 C9
      :    44 2E 18 A7 1E 77 5F B1 E7 FF 74 DB 04 9C AC 00
      :    61 A8 70 F8 B7 05 60 4B D3 79 B4 88 53 AE 90 D0
      :    11
      :    }
      :  }
    :  }
  :  }
:  }
```

```
PrivateKeyInfo ::= SEQUENCE {
    version      Version,
    privateKeyAlgorithm AlgorithmIdentifier,
    privateKey    OCTET STRING ::= ECPriateKey,
}
```

```
AlgorithmIdentifier ::= SEQUENCE {
    algorithm    OBJECT IDENTIFIER ::= id-ecPublicKey,
    parameters   ECPParameters
}
```

```
ECPParameters ::= CHOICE {
    namedCurve      OBJECT IDENTIFIER
    -- implicitCurve NULL
    -- specifiedCurve SpecifiedECDomain
}
```

```
ECPriateKey ::= SEQUENCE {
    version      INTEGER { ecPrivkeyVer1(1) } (ecPrivkeyVer1),
    privateKey    OCTET STRING,
    parameters    [0] ECPParameters {{ NamedCurve }} OPTIONAL,
    publicKey     [1] BIT STRING OPTIONAL
}
```

<https://tools.ietf.org/html/rfc5915>

<https://tools.ietf.org/html/rfc5208>

EC public key file format

```
$ openssl ec -in priv.pem -pubout -out pub.pem
$ cat pub.pem
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEo5fWms2aid56D6q9XLD5QAQJjvG8
i00P028akFtaIvVSkZ9EG0o1eG1Bt43dCSXSGaeMdUmgZ1PrYEEPnKxi0w==
-----END PUBLIC KEY-----
```

```
$ openssl ec -in pub.pem -pubin -outform der --out pub.der
$ dumpasn1 pub.der
0 89: SEQUENCE {
2 19: SEQUENCE {
4 7: OBJECT IDENTIFIER ecPublicKey (1 2 840 10045 2 1)
13 8: OBJECT IDENTIFIER prime256v1 (1 2 840 10045 3 1 7)
: }
23 66: BIT STRING
: 04 A3 97 D6 9A CD 9A 89 DE 7A OF AA BD 5C B0 F9
: 40 04 09 8E F1 BC 8B 43 8F 3B 6F 1A 90 5B 5A 22
: F5 52 91 9F 44 18 EA 35 78 69 41 B7 8D DD 09 25
: D2 19 A7 8C 75 49 A0 66 53 EB 60 41 OF 9C AC 62
: D3
: }
```

```
SubjectPublicKeyInfo ::= SEQUENCE {
    algorithm      AlgorithmIdentifier,
    subjectPublicKey BIT STRING ::= ECPublicKey
}
```

ECPublicKey ::= OCTET STRING

```
AlgorithmIdentifier ::= SEQUENCE {
    algorithm OBJECT IDENTIFIER ::= id-ecPublicKey,
    parameters ECPParameters
}
```

id-ecPublicKey OBJECT IDENTIFIER ::= {
iso(1) member-body(2) us(840) ansi-X9-62(10045) keyType(2) 1 }

```
ECPParameters ::= CHOICE {
    namedCurve      OBJECT IDENTIFIER
    -- implicitCurve NULL
    -- specifiedCurve SpecifiedECDomain
}
```

<https://tools.ietf.org/html/rfc5480>

EC point compression

```
$ openssl ec -in pub.pem -pubin -conv_form compressed -outform der -out pub_compressed.der
$ dumpasn1 pub_compressed.der
0 57: SEQUENCE {
2 19: SEQUENCE {
4 7: OBJECT IDENTIFIER ecPublicKey (1 2 840 10045 2 1)
13 8: OBJECT IDENTIFIER prime256v1 (1 2 840 10045 3 1 7)
: }
23 34: BIT STRING
: 03 A3 97 D6 9A CD 9A 89 DE 7A 0F AA BD 5C B0 F9
: 40 04 09 8E F1 BC 8B 43 8F 3B 6F 1A 90 5B 5A 22
: F5
: }
```

- The first byte of `ECPublicKey` indicates:
 - 0x04 – uncompressed form
 - 0x03 – compressed form (y is odd)
 - 0x02 – compressed form (y is even)
 - 0x00 – point at infinity
- Patent expired in 2018

Elliptic Curve Digital Signature Algorithm (ECDSA)

Signing (given hash h , EC private key d):

1. Generate a random nonce k in the range $[1, n - 1]$
 2. Calculate the random point $R = k \times G$ and set r to its x coordinate: $r = R.x$
 3. Calculate modular inverse k^{-1} , such that $k \cdot k^{-1} \equiv 1 \pmod n$
 4. Calculate $s = k^{-1} \cdot (h + r \cdot d) \pmod n$ *Restart if $s = 0$*
 5. Return the signature r, s
- Bit-length of h must not exceed the bit-length of the curve's order n
 - Longer hash values must be truncated to $n.bit_length()$ most significant bits
 - Shorter hash values must be left-padded with zeros
 - Nonce k must be secret and random(!)
 - Deterministic ECDSA (RFC 6979): nonce k derived using HMAC from h and d

Verification (given hash h , signature r, s and EC public key Q): *Validate Q*
Verify r and s in $[1, n - 1]$

1. Calculate $R' = (h \cdot s^{-1}) \times G + (r \cdot s^{-1}) \times Q$
2. Verification successful if $R'.x = r \pmod n$

Task: ECDSA utility

Implement an ECDSA signing and verification utility.

```
$ ./ecdsa.py
```

Usage:

```
sign <private key file> <file to sign> <signature output file>
```

```
verify <public key file> <signature file> <file to verify>
```

```
$ ./ecdsa.py sign priv.pem filetosign signature
```

```
$ dumpasn1 signature
```

```
0 69: SEQUENCE {
```

```
2 33: INTEGER
```

```
: 00 E6 52 A8 B4 46 61 E9 2A 35 CD 1B 11 A7 01 5E
```

```
: FA OD 2D 95 D3 56 C5 18 BC F8 72 28 4B 70 DA E3
```

```
: F7
```

```
37 32: INTEGER
```

```
: 54 B0 D3 BF 29 77 B9 89 79 E3 64 04 24 9A C4 C3
```

```
: 15 45 93 1C AC 20 71 A0 59 50 F4 BD E8 DE 11 A5
```

```
: }
```

```
ECDSA-Sig-Value ::= SEQUENCE {  
    r  INTEGER,  
    s  INTEGER  
}
```

<https://tools.ietf.org/html/rfc5480>

- Use the NIST P-256 (secp256r1) curve and SHA384
- Implement ECDSA yourself using EC point operations
- Modular inverse $x \bmod n$ can be calculated using `pow(x, -1, n)`

Task: Test cases

```
#!/bin/bash
echo "[+] Generating EC key pair..."
openssl genpkey -algorithm EC -pkeyopt ec_paramgen_curve:prime256v1 -out priv.pem
openssl ec -in priv.pem -pubout -out pub.pem

echo "[+] Testing ECDSA signing..."
dd if=/dev/urandom of=filetosign bs=1M count=1
./ecdsa.py sign priv.pem filetosign signature
openssl dgst -sha384 -verify pub.pem -signature signature filetosign
Verified OK

echo "[+] Testing ECDSA verification..."
openssl dgst -sha384 -sign priv.pem -out signature filetosign
./ecdsa.py verify pub.pem signature filetosign
Verified OK

echo "[+] Testing ECDSA failed verification..."
openssl dgst -sha1 -sign priv.pem -out signature filetosign
./ecdsa.py verify pub.pem signature filetosign
Verification failure
```

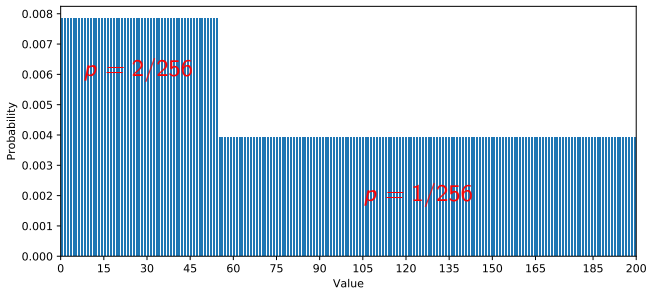
EC point operations in Python

```
$ python3
>>> from secp256r1 import curve
>>> curve.g
[48439561293906451759052585252797914202762949526041747995844080717082404635286,
 36134250956749795798585127919587881956611106672985015071877198253568414405109]
>>> curve.g[0]
48439561293906451759052585252797914202762949526041747995844080717082404635286
>>> curve.mul(curve.g, 5)
[36794669340896883012101473439538929759152396476648692591795318194054580155373,
 101659946828913883886577915207667153874746613498030835602133042203824767462820]
>>> curve.n
115792089210356248762697446949407573529996955224135760342422259061068512044369
>>> curve.mul(curve.g, curve.n)
[None, None]
>>> curve.mul(curve.g, curve.n+5)
[36794669340896883012101473439538929759152396476648692591795318194054580155373,
 101659946828913883886577915207667153874746613498030835602133042203824767462820]
>>> curve.add(curve.g, curve.g)
[56515219790691171413109057904011688695424810155802929973526481321309856242040,
 3377031843712258259223711451491452598088675519751548567112458094635497583569]
>>> curve.mul(curve.g, 2)
[56515219790691171413109057904011688695424810155802929973526481321309856242040,
 3377031843712258259223711451491452598088675519751548567112458094635497583569]
>>> curve.valid(curve.g)
True
>>> curve.valid([1,5])
False
>>> curve.compress(curve.g).hex()
'036b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296'
>>> curve.decompress(curve.compress(curve.g))
[48439561293906451759052585252797914202762949526041747995844080717082404635286,
 36134250956749795798585127919587881956611106672985015071877198253568414405109]
```

Generate random number in given range

Generate a random integer in range $[0, 200]$:

- `bi(os.urandom(1)) % 201`



- Decrease bias: `bi(os.urandom(64)) % 201`
- Rejection sampling: regenerate until in range

Questions

- What is the Elliptic Curve Discrete Logarithm Problem?
- What constitutes a private key in ECC?
- What constitutes a public key in ECC?
- What is elliptic curve point multiplication?
- What is point validation and why is it needed in ECDH?
- What does 256 bits for a 256-bit curve denote?
- Why is ECC preferred over RSA?
- How can data be encrypted using ECC?
- What is the largest value that can be signed directly using ECDSA?