Exercises: elliptic-curve based cryptography

Jean-Sébastien Coron

Université du Luxembourg

For the exercises below, install the Sage library, available at `http://www.sagemath.org/`. Alternatively, you can use the Sage Cell Server at `https://sagecell.sagemath.org`. Please submit a `.ipynb` file.

1 NIST Curve P-192

The following elliptic-curve of equation:

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

is defined in [1], with:

$$\begin{aligned} p &= 6277101735386680763835789423207666416083908700390324961279 \\ n &= 6277101735386680763835789423176059013767194773182842284081 \\ b &= \texttt{0x64210519e59c80e70fa7e9ab72243049feb8deecc146b9b1} \\ G_x &= \texttt{0x188da80eb03090f67cbf20eb43a18800f4ff0afd82ff1012} \\ G_y &= \texttt{0x07192b95ffc8da78631011ed6b24cdd573f977a11e794811} \end{aligned}$$

where n is the group order and $G = (G_x, G_y)$ is a generator of this group.

1. Verify that p and n are primes, and that G belongs to the curve.

```
def NIST_P192():
  p=6277101735386680763835789423207666416083908700390324961279
  n=6277101735386680763835789423176059013767194773182842284081
  b=0x64210519e59c80e70fa7e9ab72243049feb8deecc146b9b1
  G=(0x188da80eb03090f67cbf20eb43a18800f4ff0afd82ff1012, \
     0x07192b95ffc8da78631011ed6b24cdd573f977a11e794811)
  a=mod(-3,p)
  b=mod(b,p)
  G=(mod(G[0],p),mod(G[1],p))
  return p,n,a,b,G

def inCurve(P,a,b):
  return P[1]^2==P[0]^3+a*P[0]+b
```

2. Implement the elliptic curve addition algorithm. The point at infinity $\mathcal{O}$ can be represented as $(0, 0)$.

```
def add(P,Q,a):
  pass
```

3. Implement the double and add algorithm

```
def doubleAdd(P,e,a):
  pass
```

4. Verify that $nG = \mathcal{O}$. Therefore G is a generator of the elliptic-curve group of prime order n.

```
def test_NIST_P192():
  p,n,a,b,G=NIST_P192()
  assert inCurve(G,a,b)
  assert doubleAdd(G,n,a)==(0,0)
```

2 EC El-Gamal Encryption

The goal is to implement El-Gamal encryption and decryption over the NIST curve P-192. Let $\mathbb{G}$ be an elliptic curve subgroup of prime order n and G a generator of $\mathbb{G}$. The public key is (G, T) where $T = \alpha \cdot G$, where $\alpha \xleftarrow{R} \mathbb{Z}_n$. The private-key is α.

We consider a variant of El-Gamal for a message $m \in \{0,1\}^k$, using a hash function $H : \{0,1\}^* \to \{0,1\}^k$, with the ciphertext:

$$c = (r \cdot G, H(r \cdot T) \oplus m)$$

We can use the SHA1 hash function, with $k = 160$:

```
import hashlib

def sha1(m):
  h=hashlib.sha1()
  h.update(m.encode("utf-8"))
  return h.hexdigest()
```

To convert an integer into a string, you can use the str function. To compute the xor between two strings, you can use the function xor_string below. To pad a string to 160 bits, you can use the pad_to_160_bits function below.

```
def xor_strings(s1, s2):
  return ''.join(chr(ord(a) ^^ ord(b)) for a, b in zip(s1, s2))

def pad_to_160_bits(s):
  return s.ljust(20, '\0')
```

Implement the El-Gamal encryption scheme, with key generation, encryption and decryption.

```
def ElGamalECCKeyGen():
  p,n,a,b,G=NIST_P192()
  pass

def ElGamalECCEncrypt(m, G, T, a, n):
  pass

def ElGamalECCDecrypt(c1, c2, a, alpha):
  pass
```

Check that decryption works.

```
def testElGamalECC():
  G,T,a,n,alpha=ElGamalECCKeyGen()
  m="Hello"
  mpad=pad_to_160_bits(m)
  c1,c2=ElGamalECCEncrypt(mpad,G,T,a,n)
  m2=ElGamalECCDecrypt(c1,c2,a,alpha)
  assert m2==mpad
  print(m2)
```

References

1. FIPS PUB 186-3, *Digital Signature Standard (DSS)*. Available at `http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf`