Lab 1

Constructions

Martin R. Albrecht, Guillaume Bonnoron and Léo Duc

22 March 2017

In this lab, we will implement elementary cryptographic primitives based on lattices, namely a public-key encryption scheme in a single bit version, multi-bit version and ring version. The key take-aways will be about random sampling in lattices, security evaluation and matrix manipulations.

Introduction

As a warm up, we ask you to use the Discrete Gaussian Sampler from Sage which follow the techniques described by Ducas et al. First import the necessary class:

\[
\text{sage: from sage.stats.distributions.discrete_gaussian.integer} \\import \text{DiscreteGaussianDistributionIntegerSampler}
\]

As its explicit name suggests, you can now generate integers from this sampler, providing it with the standard deviation \(\sigma\) of the distribution you wish to use.

**Note:** In cryptography, we commonly define a Gaussian distribution with center \(c\) and parameter \(s\) as the distribution where elements occur with a probability proportional to \(\exp(-\pi s^2 x^2)\). Sage defines a Gaussian distribution with center \(c\) and standard deviation \(\sigma\) as the distribution where elements occur with a probability proportional to \(\exp(-\frac{(x-c)^2}{2\sigma^2})\). Thus, the Gaussian width parameter \(s\) as used in cryptography relates to the standard deviation used in Sage as \(s = \sqrt{2\pi\sigma}\).

\[
\text{sage: D = DiscreteGaussianDistributionIntegerSampler(sigma=2.0, c=5)}
\]

\[
\text{sage: D, D(), D(), D(), D(), D()}
\]

\[(6, 4, 2, 6, 3, 3)\]

Unfortunately for us, Sage prefers to represent number modulo \(q\) in the range \([0, q - 1]\) whereas as cryptographers we would rather have them in \([-\lfloor q/2 \rfloor, \lfloor q/2 \rfloor]\). In order to balance things, you might wish to use the function below.

def balance(e, q=None):
    try:
        p = parent(e).change_ring(ZZ)
        return p(balance(e_, q=q) for e_ in e)
    except (TypeError, AttributeError):
        if q is None:
            try:
                q = parent(e).order()
            except AttributeError:
                q = parent(e).base_ring().order()
        return ZZ(e)-q if ZZ(e)>q/2 else ZZ(e)

Public-Key Encryption — Single Bit

The public-key encryption scheme we want to implement goes as follows. Given a dimension $n$, a modulus $q$, we work in $\mathbb{Z}_q^n = (\mathbb{Z}/q\mathbb{Z})^n$. We also need a discrete Gaussian distribution of stddev $\sigma$.

- **ppGen:** $A \in \mathbb{Z}_q^{n \times m}$ a public uniformly random matrix, with $m = 2n\lceil \log q \rceil$.

- **KeyGen:** The public key is $b^t = s^t A + e^t \mod q$, with $s \in \mathbb{Z}_q^n$ and $e \in \mathbb{Z}_q^m$ sampled from the Gaussian distribution. The secret key is $s$.

- **Enc:** To encrypt a bit $\mu$, consider $M = \begin{bmatrix} A & b^t \end{bmatrix} \in \mathbb{Z}_q^{(n+1) \times m}$ and compute
  \[
  c = M \cdot x + \begin{bmatrix} 0 \\ \mu \cdot \lceil q/2 \rceil \end{bmatrix} = \begin{bmatrix} Ax \\ b \cdot x \end{bmatrix} + \begin{bmatrix} 0 \\ \mu \cdot \lceil q/2 \rceil \end{bmatrix} \in \mathbb{Z}_q^{n+1}
  \]
  where $x$ is uniform in $\{0, 1\}^m$.

- **Dec:** To decrypt, compute:
  \[
  (-s, 1)^t \cdot c = (-s, 1)^t \cdot M \cdot x + \mu \cdot \lceil q/2 \rceil \\
  = (e^t \cdot x) + \mu \cdot \lceil q/2 \rceil \\
  \approx \mu \cdot \lceil q/2 \rceil
  \]
  If this is closer to $0$ than to $\lfloor q/2 \rfloor$ output $0$, otherwise $1$. It works if $\langle e \cdot x \rangle < q/4$ so $q$ and $\sigma$ should be chosen accordingly.

**Ex 1:** Using the Gaussian sampler from above, implement the whole scheme i.e. the public parameter and key generations, the encryption and the decryption.

Public-Key Encryption — Multi-Bit

To improve the efficiency of the encryption scheme, it is possible to encrypt multiple bits of plaintext in one ciphertext.\(^4\) In the single


bit setting, we had for 1 bit of plaintext: \( n^2 \) integers of public parameter, \( n \) integers of public key and of secret key, and \( n + 1 \) integers of ciphertext. With the multibit approach, we can encrypted \( k^2 \) bits for only a factor \( k \) expansion in the size of the keys and ciphertexts.

The generalization of the scheme is quite straightforward. The operations remain the same, the only differences is the shift from vectors of size \( n \) to matrices of dimensions \( n \times k \) for \( s, e \) and \( b \) and \( x \). Thus \( e \) becomes a \((n + k) \times k\) matrix where the bottom \( k \times k \) coefficients store the masked encryption of \( k^2 \) bits \( \mu_{i,j} \).

**Ex 2:** Adapt your previous code to handle multiple plaintext bits.

**Public-Key encryption — Ring Setting**

As a last improvement, we will now shift our scheme from the generic lattice setting where \( A \) is uniformly random in \( \mathbb{Z}_{q}^{n \times n} \) to the ring setting. For this, we will work in a polynomial ring, e.g. \( R = \mathbb{Z}_q[X]/(X^n + 1) \) where \( n \) is a power of 2 and \( q \) is prime as before.

In this setting, the scheme is adapted as follow:

- **ppGen:** \( a \in R \) a public uniformly random polynomial of \( R \).
- **KeyGen:** The public key is \( b = s \cdot a + e \in R \), with \( s, e \in R \) sampled from the Gaussian distribution. The secret key is \( s \).
- **Enc:** To encrypt a binary polynomial \( m \), pick random \( r, e', e'' \in R \) from the Gaussian distribution and compute
  \[(c_0, c_1) = (a \cdot r + e', b \cdot r + e'' + m \cdot \lfloor q/2 \rfloor)\]
- **Dec:** To decrypt, one computes:
  \[c_1 - s \cdot c_0 = m \cdot \lfloor q/2 \rfloor + b \cdot r - s \cdot a \cdot r + e'' - s \cdot e'\]
  \[= m \cdot \lfloor q/2 \rfloor + e \cdot r + e'' - s \cdot e'\]
  \[\approx m \cdot \lfloor q/2 \rfloor\]

So for each coefficient we apply the same rule as before, if it is closer to 0 than to \( \lfloor q/2 \rfloor \) output 0, otherwise 1.

**Ex 3:** For this adaptation, more code changes are needed. You can continue to use the integer sampler, but Sage comes with a polynomial sampler. The code snippet below shows you how to use polynomial ring in Sage and its Gaussian Sampler over Polynomials.

```python
from sage.stats.distributions.discrete_gaussian_polynomial import DiscreteGaussianDistributionPolynomialSampler
...
Zq = IntegerModRing(q)
```
Rq.<x> = Zq['x'].quotient_ring(x^n+1)
P = DiscreteGaussianDistributionPolynomialSampler(Rq, n, sigma)
P()

**Security Evaluation**

Now that the scheme is working and fairly efficient, the question remains of its level of security. Here we have \( n \) that determines both the modulus \( q \) and the standard deviation \( \sigma \). So we will play with \( n \), and later also with \( q \) and \( \sigma \), and explore the level of security that we obtain.

For this work, we will use the estimator\(^7\) that models the performance of (nearly) all existing attacks against LWE. The project page presents a basic use of the estimator. The core components is the `estimate_lwe()` function that computes estimated costs of several attacks (see the project page for the details on the attacks). This function takes as input the LWE parameters to assess:\( n \) the dimension, \( q \) the modulus and \( \alpha = \sqrt{2\pi} \cdot \frac{\sigma}{q} \) which captures the error width with respect to \( q \).

**Ex 4:**

1. Assess the security of your implementations above.
2. What \( n \) should you pick to have 80 bits of security? 128 bits? 256?
3. Can you adapt the choice of \( q \) and \( \sigma \) in our code to improve the security while maintaining correctness?

To get you started, try:

```sage
load("https://bitbucket.org/malb/lwe-estimator/raw/HEAD/estimator.py")
set.verbose(1)
...
_=estimate_lwe(n, alpha, q, skip="arora-gb")
```

In the code above

- we loaded the Sage Module. From a remote location like here, or from a local file;
- set verbosity to something higher than zero to get more detailed input; and
- called a highlevel function to estimate the running time of various attacks.

**Ex 5:** Compare your results with those obtained by running the estimation scripts for *A New Hope*.\(^8\)

---


\(^8\) https://github.com/tpoeppelmann/newhope/tree/master/scripts
Ex 6: Use monkey patching to modify the behaviour of the estimator. For example, replace the cost function for lattice reduction with a different cost function, say, a function which assumes only one SVP call is necessary for BKZ.

Example Solutions

Public-key Encryption — Single Bit

```python
from sage.stats.distributions.discrete_gaussian_integer import 
DiscreteGaussianDistributionIntegerSampler
class PKESingleBit(object):
    def __init__(self, dimension):
        self.n = dimension
        self.q = next_prime(self.n^2)
        self.m = 2*self.n*ceil(log(self.q, 2))
        self.sigma = sqrt(self.n/(2*pi.n()))
        self.D = DiscreteGaussianDistributionIntegerSampler(sigma=self.sigma)
        self.Zq = IntegerModRing(self.q)

def pp_gen(self):
    self.A = random_matrix(self.Zq, self.n, self.n)

def keygen(self):
    s = vector(self.Zq, [self.D() for _ in range(self.n)])
    e = vector(self.Zq, [self.D() for _ in range(self.m)])
    b = s * self.A + e
    return s, b

def encrypt(self, m, pk):
    M = self.A.stack(pk)
    x = random_vector(self.m, 0, 2)
    c = M*x
    c[self.n] = (c[self.n] + m*self.q//2) % self.q
    return c

def decrypt(self, c, sk):
    d = list(-sk)
    d.append(1)
    m.dec = balance (vector(d) * c, self.q)
    return round(m.dec * 2/self.q) % 2
dimension = 150
message = randint(0, 1)
scheme = PKESingleBit(dimension)
scheme.pp_gen()
sk, pk = scheme.keygen()
c = scheme.encrypt(message, pk)
m.dec = scheme.decrypt(c, sk)
print message
print m.dec
```

Public-Key Encryption — Multi-Bit

```python
from sage.stats.distributions.discrete_gaussian_integer import 
DiscreteGaussianDistributionIntegerSampler
class PKEMultiBit(object):
    def __init__(self, dimension, packing):
```
self.n = dimension
self.k = packing
self.q = next_prime(self.n^2)
self.sigma = sqrt(self.n)/(2*pi.n())
self.D = DiscreteGaussianDistributionIntegerSampler(sigma=self.sigma)
self.Zq = IntegerModRing(self.q)
def pp_gen(self):
    self.A = random_matrix(self.Zq, self.n, self.n)
def keygen(self):
    Zq, n, k = self.Zq, self.n, self.k
    s = matrix(Zq, n, k, [self.D() for _ in range(n*k)])
    e = matrix(Zq, n, k, [self.D() for _ in range(n*k)])
    b = s.transpose() * self.A + e.transpose()
    return s, b
def encrypt(self, m, pk):
    x = random_matrix(ZZ, self.n, self.k, x=2)
    m = zero_matrix(self.Zq, self.n, self.k).stack(m)
    M = self.A.stack(pk)
    c = (M*x + m * (self.q//2)) % self.q
    return c
def decrypt(self, c, sk):
    d = -sk.transpose()
    d = d.augment(identity_matrix(self.k))
    m_dec = balance(d * c, self.q) * 2 / self.q
    return m_dec.apply_map(lambda x: round(x)) % 2
dimension = 150
packing = 4
message = random_matrix(ZZ, packing, x=2)
scheme = PKEMultiBit(dimension, packing)
scheme.pp_gen()
sk, pk = scheme.keygen()
c = scheme.encrypt(message, pk)
m.dec = scheme.decrypt(c, sk)

print message
print m.dec

Public-Key Encryption — Ring Setting

def __init__(self, dimension):
    self.n = dimension
    self.q = next_prime(self.n^2)
    self.sigma = sqrt(self.n)/(2*pi.n())
    Zq = IntegerModRing(self.q)
    Pq.<y> = Zq['y']
    self.Rq = Pq.quotient_ring(y^dimension+1)
    self.P = DGSPolySampler(self.Rq, self.n, self.sigma)
def pp_gen(self):
    self.a = self.Rq.random_element()

def keygen(self):
    s = self.P()
    e = self.P()
    b = s * self.a + e
    return s, b

from sage.stats.distributions.discrete_gaussian_polynomial import 
DiscreteGaussianDistributionPolynomialSampler as DGSPolySampler
class PKERing(object):
    def __init__(self, dimension):
        self.n = dimension
        self.q = next_prime(self.n^2)
        self.sigma = sqrt(self.n)/(2*pi.n())
        Zq = IntegerModRing(self.q)
        Pq.<y> = Zq['y']
        self.Rq = Pq.quotient_ring(y^dimension+1)
        self.P = DGSPolySampler(self.Rq, self.n, self.sigma)
    def pp_gen(self):
        self.a = self.Rq.random_element()
```python
def encrypt(self, m, pk):
    r = self.P()
    c = (self.a + r + self.P(), pk + r + self.P() + self.Rq(m) + (self.q//2))
    return c

def decrypt(self, c, sk):
    m_dec = ([c[1] - sk * c[0]].list()
    return map(lambda x: round(2/self.q * balance(x, self.q)) % 2, m_dec)

dimension = 16
message = [randint(0, 1) for _ in range(dimension)]
scheme = PKERing(dimension)
scheme.pp_gen()
sc, pk = scheme.keygen()
c = scheme.encrypt(message, pk)
m.dec = scheme.decrypt(c, sc)

print message
print m.dec

Security Evaluation — Monkey Patching

First download estimator.py, then try:

```python
import estimator as est
def bkz_runtime_k_sieve_bdgl16_asymptotic_1(k, n):
    return est.bkz_runtime_k_sieve_bdgl16_asymptotic(k, 1)
bkz_estimate = bkz_runtime_k_sieve_bdgl16_asymptotic_1
args = {"optimisation_target":"sieve"}

try:
est.bkz_runtime_k_sieve_asymptotic, bkz_estimate = bkz_estimate, est.bkz_runtime_k_sieve_asymptotic
print est.cost_str(est.sis(n, alpha, q, **args))
finally:
est.bkz_runtime_k_sieve_asymptotic, bkz_estimate = bkz_estimate, est.bkz_runtime_k_sieve_asymptotic
```