Lab 1

Constructions

Martin R. Albrecht, Guillaume Bonnoron and Léo Ducas

22 March 2017

In this lab, we will implement elementary cryptographic primitives based on lattices, namely a public-key encryption scheme[1] in a single bit version, multi-bit version and ring version[2]. The key take-aways will be about random sampling in lattices, security evaluation and matrix manipulations.

[1] Oded Regev. "On lattices, learning with errors, random linear codes, and cryptography". In: *37th ACM STOC*. ed. by Harold N. Gabow and Ronald Fagin. ACM Press, May 2005, pp. 84–93. DOI: 10.1145/1060590.1060603.

[2] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. "On Ideal Lattices and Learning with Errors over Rings". In: *EUROCRYPT 2010*. Ed. by Henri Gilbert. Vol. 6110. LNCS. Springer, Heidelberg, 2010, pp. 1–23. DOI: 10.1007/978-3-642-13190-5_1.

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.[3] First import the necessary class:

[3] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. "Lattice Signatures and Bimodal Gaussians". In: *CRYPTO 2013, Part I*. ed. by Ran Canetti and Juan A. Garay. Vol. 8042. LNCS. Springer, Heidelberg, Aug. 2013, pp. 40–56. DOI: 10.1007/978-3-642-40041-4_3.

```
sage: from sage.stats.distributions.discrete_gaussian_integer \
import DiscreteGaussianDistributionIntegerSampler
```

As its explicit name suggests, you can now generate integers from this sampler, providing it with the standard deviation σ 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 $\exp(-\pi\frac{(x-c)^2}{s^2})$. Sage defines a Gaussian distribution with center c and standard deviation σ 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 $s = \sqrt{2\pi}\sigma$.

```
sage: D = DiscreteGaussianDistributionIntegerSampler(sigma=2.0, c=5)
sage: D
Discrete Gaussian sampler over the Integers with sigma = 2.0 and c = 5
```

This creates the sampler that you can then call to produce your integers. Here they are centered around 5 and have standard deviation 2.

```
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 $[\lceil -q/2 \rceil, \lfloor q/2 \rfloor]$. 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 σ.

- ppGen: $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$ a public uniformly random matrix, with $m = 2n\lceil \log q \rceil$.
- KeyGen: The public key is $\mathbf{b}^t = \mathbf{s}^t\mathbf{A} + \mathbf{e}^t \mod q$, with $\mathbf{s} \in \mathbb{Z}_q^n$ and $\mathbf{e} \in \mathbb{Z}_q^m$ sampled from the Gaussian distribution. The secret key is $\mathbf{s}$.
- Enc: To encrypt a bit μ, consider $\mathbf{M} = \left[\begin{smallmatrix}\mathbf{A}\\ \mathbf{b}^t\end{smallmatrix}\right] \in \mathbb{Z}_q^{(n+1)\times m}$ and compute

$$\begin{aligned}\mathbf{c} &= \mathbf{M} \cdot \mathbf{x} + \left(\begin{smallmatrix}\mathbf{0}\\ \mu \cdot \lfloor q/2 \rceil\end{smallmatrix}\right)\\ &= \left(\begin{smallmatrix}\mathbf{A}\cdot\mathbf{x}\\ \langle \mathbf{b}\cdot\mathbf{x}\rangle\end{smallmatrix}\right) + \left(\begin{smallmatrix}\mathbf{0}\\ \mu \cdot \lfloor q/2 \rceil\end{smallmatrix}\right) \in \mathbb{Z}_q^{n+1}\end{aligned}$$

  where $\mathbf{x}$ is uniform in $\{0,1\}^m$.
- Dec: To decrypt, compute:

$$\begin{aligned}(-\mathbf{s}, 1)^t \cdot \mathbf{c} &= (-\mathbf{s}, 1)^t \cdot \mathbf{M} \cdot \mathbf{x} + \mu \cdot \lfloor q/2 \rceil\\ &= \langle \mathbf{e}^t \cdot \mathbf{x} \rangle + \mu \cdot \lfloor q/2 \rceil\\ &\approx \mu \cdot \lfloor q/2 \rceil\end{aligned}$$

  If this is closer to 0 than to $\lfloor q/2 \rceil$ output 0, otherwise 1. It works if $\langle \mathbf{e} \cdot \mathbf{x} \rangle < q/4$ so q and σ 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.[4,5,6] 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 $\mathbf{s}, \mathbf{e}$ and $\mathbf{b}$ and $\mathbf{x}$. Thus $\mathbf{c}$ 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.

[4] Richard Lindner and Chris Peikert. "Better Key Sizes (and Attacks) for LWE-Based Encryption". In: *CT-RSA 2011*. Ed. by Aggelos Kiayias. Vol. 6558. LNCS. Springer, Heidelberg, Feb. 2011, pp. 319–339. DOI: 10.1007/978-3-642-19074-2_21.

[5] Jintai Ding. *New cryptographic constructions using generalized learning with errors problem*. Cryptology ePrint Archive, Report 2012/387. http://eprint.iacr.org/2012/387. 2012.

[6] Joppe W. Bos et al. "Frodo: Take off the Ring! Practical, Quantum-Secure Key Exchange from LWE". in: *ACM CCS 16*. Ed. by Edgar R. Weippl et al. ACM Press, Oct. 2016, pp. 1006–1018. DOI: 10.1145/2976749.2978425.

Public-Key encryption — Ring Setting

As a last improvement, we will now shift our scheme from the generic lattice setting where $\mathbf{A}$ 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: $\mathbf{a} \in R$ a public uniformly random polynomial of R.
- KeyGen: The public key is $\mathbf{b} = \mathbf{s} \cdot \mathbf{a} + \mathbf{e} \in R$, with $\mathbf{s}, \mathbf{e} \in R$ sampled from the Gaussian distribution. The secret key is $\mathbf{s}$.
- Enc: To encrypt a binary polynomial $\mathbf{m}$, pick random $\mathbf{r}, \mathbf{e}', \mathbf{e}'' \in R$ from the Gaussian distribution and compute

$$(\mathbf{c}_0, \mathbf{c}_1) = (\mathbf{a} \cdot \mathbf{r} + \mathbf{e}', \mathbf{b} \cdot \mathbf{r} + \mathbf{e}'' + \mathbf{m} \cdot \lfloor q/2 \rceil)$$

- Dec: To decrypt, one computes:

$$\begin{aligned} \mathbf{c}_1 - s \cdot \mathbf{c}_0 &= \mathbf{m} \cdot \lfloor q/2 \rceil + \mathbf{b} \cdot \mathbf{r} - \mathbf{s} \cdot \mathbf{a} \cdot \mathbf{r} + \mathbf{e}'' - \mathbf{s} \cdot \mathbf{e}' \\ &= \mathbf{m} \cdot \lfloor q/2 \rceil + \mathbf{e} \cdot \mathbf{r} + \mathbf{e}'' - \mathbf{s} \cdot \mathbf{e}' \\ &\approx \mathbf{m} \cdot \lfloor q/2 \rceil \end{aligned}$$

  So for each coefficient we apply the same rule as before, if it is closer to 0 than to $\lfloor q/2 \rceil$ 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 Gaussiam Sampler over Polynomials.

```
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 σ. So we will play with n, and later also with q and σ, 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.

[7] Martin R. Albrecht, Rachel Player, and Sam Scott. *On The Concrete Hardness Of Learning With Errors.* Cryptology ePrint Archive, Report 2015/046. http://eprint.iacr.org/2015/046. 2015.

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 σ 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")
sage: set_verbose(1)
...
sage: _ = 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]

Ex 6: Use monkey patching to modify the behaviour of the estimator.[9] 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.[10]

[8] https://github.com/tpoeppelmann/newhope/tree/master/scripts

[9] https://stackoverflow.com/questions/5626193/what-is-a-monkey-patch

[10] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. *Post-quantum key exchange - a new hope*. Cryptology ePrint Archive, Report 2015/1092. http://eprint.iacr.org/2015/1092. 2015.

Example Solutions

Public-key Encryption — Single Bit

```
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

```
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

```
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()

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

  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()
sk, pk = scheme.keygen()
c = scheme.encrypt(message, pk)
m_dec = scheme.decrypt(c, sk)

print message
print m_dec
```

Security Evaluation — Monkey Patching

First download `estimator.py`, then try:

```
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
```