Lattice-Based Cryptography and the LWE Cryptosystem

Ben Young

March 23, 2020

Contents

1 Abstract

This report provides an overview of mathematical lattices and lattice-based cryptography and provides an in-depth exploration into and implementation of one particular cryptosystem, the Learning With Errors (LWE) cryptosystem.

A lattice is a structured, periodic set of points in n-dimensional space. It is generated as the set of all integer-coefficient linear combinations of a set of n-length basis vectors. Lattices are promising cryptographic structures because they form the basis for several problems that are conjectured to be unsolvable in polynomial time for both classical and quantum algorithms. One such problem is the shortest vector problem (SVP), which entails finding the shortest vector in a lattice to within a certain approximation factor. Another (at first seemingly unrelated to lattices) problem conjectured to be hard if the learning with errors problem, which entails determining whether a vector has been sampled from a uniform distribution or a rounded normal distribution. Worst-case SVP can be reduced to LWE. We also give an overview and an implementation of a public-key cryptosystem that is secure in that successfully attacking it implies having found a solution to the LWE problem.

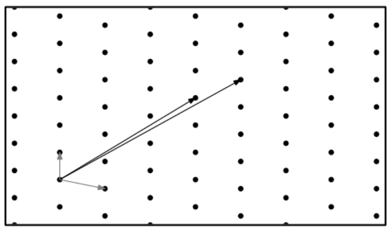
2 Introduction: Lattices

A *lattice* is essentially the set of all integer-coefficient linear combinations of a given set of n-length basis vectors, forming a grid-like pattern in n-dimensional space. More formally, if $\mathbf{b}_1, \cdots, \mathbf{b}_n \in \mathbb{R}^n$ are n linearly independent basis vectors we can define a lattice $\mathcal{L}$ as

$$\mathcal{L}(\mathbf{b}_1, \cdots, \mathbf{b}_n) = \left\{ \sum_{i=1}^{n} x_i \mathbf{b}_i : x_i \in \mathbb{Z} \right\}$$

We usually represent the basis as a matrix

Figure 2.1: An example of a 2-dimensional lattice with basis vectors [1]



$$\mathbf{B} = [\mathbf{b_1}, cdots, \mathbf{b_n}] \in \mathbb{R}^{n \times n}$$

so that the lattice generated by $\mathbf{B}$ is

$$\mathcal{L}(\mathbf{B}) = \{\mathbf{Bx} : \mathbf{x} \in \mathbb{Z}^n\}.$$

We usually consider *q-ary lattices*, which are lattices whose member vectors are integers determined component-wise mod q - that is the are elements of $\mathbb{Z}_q^n$.

Alternatively, if we wish to construct a lattice from a matrix, instead of using its columns as the basis vectors we can use its rows to generate the lattice. This method has the advantage that we can construct lattice from non-square matrices. Given $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$, define

$$\Lambda_q(\mathbf{A}) := \{\mathbf{x} \in \mathbb{Z}^m : \mathbf{x} = \mathbf{A}^T \mathbf{s} \pmod q \text{ for some } \mathbf{s} \in \mathbb{Z}^n\} \tag{2.1}$$

Before describing Learning With Errors, we first describe two traditional lattice problems that can be reduced to it. First is the Shortest Vector Problem (SVP), formulated as follows: Given a basis $\mathbf{B}$ and approximation factor γ, find a vector $\mathbf{x}$ such that $||\mathbf{x}|| \leqslant \gamma ||\mathbf{s}||$, where $\mathbf{s}$ is the shortest nonzero vector in $\mathcal{L}(\mathbf{B})$.

Next is the Shortest Independent Vector Problem (SIVP): Given basis $\mathbf{B}$ and approximation factor γ, find n linearly independent lattice vectors $\{\mathbf{x}_1, \cdots, \mathbf{x}_n\} \subseteq \mathcal{L}(\mathbf{B})$ such that $\max_i ||\mathbf{x}_i|| \leqslant \gamma S$, where $S = \max_i ||\mathbf{x}_i||$ for the set of n linearly independent vectors $\{\mathbf{x}_1, \cdots, \mathbf{x}_n\}$ giving the smallest such quantity S (both problems involve finding approximations to the optimal solution to the problem).

Existing algorithms to solve these two problems either run in polynomial time but only for exponential approximation factors γ, or can approximate the solution to within polynomial factors but can only achieve exponential run time. Thus [1] conjectures that there is no polynomial time classical or quantum algorithm that approximates these lattice problems to within polynomial factors.

3 The LWE-Based Cryptosystem

Now we introduce the Learning With Errors (LWE)-based cryptosystem first proposed by Regev in [2]. We first describe the LWE problem. The problem has integer parameters n, m, and q. It also uses a probability distribution χ on $\mathbb{Z}_q$, which is usually a 'rounded' normal distribution centered at 0 where the sample is rounded to the nearest integer. The problem's inputs are a matrix $\mathbf{A} \in \mathbb{Z}_q^{m\times n}$ whose elements are chosen uniformly and indepentently from $\mathbb{Z}_q$ and a vector $\mathbf{v}$ either (unknown to the problem solver) chosen uniformly from $\mathbb{Z}_q^m$ or as $\mathbf{v} = \mathbf{A}\mathbf{s} + \mathbf{e}$ for uniformly chosen $\mathbf{s} \in \mathbb{Z}_q^n$ and $\mathbf{e} \in \mathbb{Z}_q^m$ sampled from χ^m. Solving the problem then consists of determining which of these two methods was used to generate $\mathbf{v}$.

The initial connection between this problem and lattice problems is that the case when $\mathbf{v} = \mathbf{A}\mathbf{s} + \mathbf{e}$ is equivalent to choosing $\mathbf{v}$ as a random point in the lattice $\Lambda_q(\mathbf{A}^T)$ and then randomly perturbing each of the point's coordinates using χ^m.

A deeper connection is established by a theorem proven in [2] and restated in [1]. Both this theorem and the following LWE-based cryptosystem use the probability distribution Ψ_α on $\mathbb{Z}_q$ defined as a normal distribution with mean 0 and standard deviation $\frac{\alpha q}{\sqrt{2\pi}}$ rounded to the nearest integer and reduced mod q. The theorem states that if we have an oracle that solves the LWE problem using distribution Ψ_α and parameters n, m, and q with $\alpha q > \sqrt{n}$, q prime and q and m smaller but not exponentially smaller than n, then there is a polynomial-time (in n) quantum algorithm that can solve reasonably-sized worst-case SVP and SIVP problems in any n-dimensional lattice. (I'd like to give at least an outline of the proof of this theorem in the final draft).

Now we can discussed the LWE-based cryptosystem proposed by [2]. [3] created a more efficient version of the cryptosystem, which is what [1] presents and what we will discuss here.

The system has integer parameters n, m, l, t, r, and q, where n is the size of the lattice underlying the system (and is thus the most important security parameter), l is the length of each message, and t is the size of the field over which the message bits are chosen (usually 2). It also has real-valued parameter $\alpha > 0$, which is the spread of the distribution Ψ_α.

The system uses function $f : \mathbb{Z}_t^l \to \mathbb{Z}_q^l$ that converts mod-t vectors to mod-q vectors by multiplying each coordinate in its input by $\frac{q}{t}$ and rounding to the nearest integer. f has corresponding inverse $f^{-1} : \mathbb{Z}_q^l \to \mathbb{Z}_t^l$ that converts mod-q vectors to mod-t vectors by multiplying each coordinate by $\frac{t}{q}$ and rounding to the nearest integer.

The private key is a random uniformly-chosen matrix $\mathbf{S} \in \mathbb{Z}_q^{n\times l}$. To determine a public key, choose two more random matrices: $\mathbf{A} \in \mathbb{Z}_q^{m\times n}$ from a uniform distribution and $\mathbf{E} \in \mathbb{Z}_q^{m\times l}$ with each entry chosen from Ψ_α. Then calculate

$$\mathbf{P} = \mathbf{A}\mathbf{S} + \mathbf{E} \in \mathbb{Z}_q^{m\times l} \tag{3.1}$$

and the public key is the pair $(\mathbf{A}, \mathbf{P})$.

To encrypt a message $\mathbf{v} \in \mathbb{Z}_t^l$ choose $\mathbf{a} \in \{-r, -r+1, \cdots, r\}^m$ uniformly at random, calculate

$$\mathbf{u} = \mathbf{A}^T\mathbf{a} \in \mathbb{Z}_q^n, \quad \mathbf{c} = \mathbf{P}^T\mathbf{a} + f(\mathbf{v}) \in \mathbb{Z}_q^l \tag{3.2}$$

and send ciphertext $(\mathbf{u}, \mathbf{c})$.

Then to decrypt a ciphertext, calculate

$$m = f^{-1}(\mathbf{c} - \mathbf{S}^T\mathbf{u}) \in \mathbb{Z}_t^l \tag{3.3}$$

and m should be the original plaintext.

[2] and [3] prove that this system is secure against known plaintext attacks given the right choice of parameters. They first show that determining whether the public key $(\mathbf{A}, \mathbf{P})$ was generated by the cryptosystem using Ψ_α or uniformly at random is equivalent to solving a LWE problem. They then show that if one could gain information using a known plaintext attack, then they would be able to distinguish between uniformly randomly generated public keys and public keys generated by the cryptosystem, so they could

solve LWE, which is a hard problem (this is another proof I want to read thoroughly and present here in a more detailed form).

3.1 Implementation

Section 5 (the Appendix) contains a rough Python implementation of this cryptosystem. The method `send_ciphertext` takes a message as a text string, converts the text string to a binary string using ASCII codes, encrypts the binary string in chunks using the above procedure, and outputs the encrypted chunks. Then `decrypt_ciphertext` decrypts the ciphertexts, converts them back from binary strings to text strings, and outputs the result. The rounding carried out by the functions f and f^{-1} used in the encryption and decryption procedures occasionally produces errors and causes the decrypted message not to match the original plaintext. Thus the program uses a simple Hamming error correcting code to correct at most one incorrect bit per message (another thing I'd like to change is to implement a more sophisticated error correcting code that can catch more than one error, because some still show up even though I'm using a fairly short message length).

4 References

[1] D. Micciancio and O. Regev. Lattice-based Cryptography. 2008. cims.nyu.edu/∼ regev/papers/pqc.pdf
[2] O. Regev, On lattices, learning with errors, random linear codes, and cryptography. In *Proc. 37th ACM Symp. on Theory of Computing*, pages 84-93, 2005. https://cims.nyu.edu/ regev/papers/qcrypto.pdf
[3] C. Peikert, V. Vaikuntanathan, and B. Waters. A framework for efficient and composable oblivious transfer. In *Advances in Cryptology*. Springer, 2008. https://eprint.iacr.org/2007/348.pdf

5 Appendix: Implementation of the LWE-based Cryptosystem

```
import secrets
import numpy as np
import math
class LWECryptosystem():
    def __init__(self, n, l, m, q, r, t, alpha):
        self.n = n
        self.l = l
        self.m = m
        self.q = q
        self.r = r
        self.t = t
        self.alpha = alpha

        #private key
        self.S = np.array([[secrets.randbelow(q) for _ in range(l)] for _ in range(n)])

        #public key
        self.A = np.array([[secrets.randbelow(q) for _ in range(n)] for _ in range(m)])
        E = np.array([[self.sample_psi() for _ in range(l)]
                      for _ in range(m)])
        self.P = np.matmul(self.A, self.S) + E

    def encrypt(self, v: np.array):
        """encrypt v: l-length vector mod t to output (u,c) where u and c are n and l leng
        vectors, respectively, mod q"""
        rand_range = list(range(-self.r, self.r+1))
        a = np.array([secrets.choice(rand_range) for _ in range(self.m)])
        return (np.matmul(np.transpose(self.A), a),
                np.matmul(np.transpose(self.P), a) + self.f(v))

    def decrypt(self, u: np.array, c: np.array):
        """u: length-n vector mod q
           c: length-l vector mod q (u and c are the two outputs of encrypt)
           return: length-l vector mod t (the original message)"""
        return self.f_inv(c - np.matmul(np.transpose(self.S), u))

    def sample_psi(self):
        """sample from normal distribution with mean 0 and std. dev. alpha*q/sqrt(2pi),
        round to nearest integer and reduce mod q"""
        return int(np.round(np.random.normal(0, self.alpha*(self.q)/math.sqrt(2*math.pi)))

    def f(self, v: np.array):
        """v: l-length vector mod q
        output: l-length vector mod t obtained by multiplying each coordinate of v by
        q/t and rounding to the nearest integer"""
        return np.round(v*(self.q/self.t)).astype('int') % self.q

    def f_inv(self, v: np.array):
        """v: l-length vector mod t
        output: l-length vector mod q obtained by multiplying each coordinate of v by
        t/q and rounding to the nearest integer"""
        return np.round(v*(self.t/self.q)).astype('int') % self.t


def send_ciphertext(message: str, system: LWECryptosystem, hamming_encoder: np.array):
    n = system.n
    segment_bits = n//2 + 1
    #each character has 7-bit ascii code
    segment_chars = segment_bits // 7

    message_segments = [message[segment_chars*i:segment_chars*(i+1)]
                        for i in range(0, len(message)//segment_chars+1)]
    ciphertexts = []
    for segment in message_segments:
        plaintext = []
        #convert to binary representation of ascii codes
        for char in segment:
            ascii_binary = '{0:b}'.format(ord(char))
            #pad front with zeros to get to length 7
            ascii_binary = '0'*(7-len(ascii_binary)) + ascii_binary
            plaintext.extend(list(ascii_binary))
        for _ in range(len(segment), segment_chars):
            #pad with spaces if number of characters is below segment_chars
            plaintext.extend(list('0100000'))
        plaintext = np.array(plaintext).astype('int')
        #add bits to end of plaintext with hammond encoder
        encoded_plaintext = np.matmul(plaintext, hamming_encoder).astype('int') % 2

        #encrypt hammond-encoded bits with cryptosystem
        u, c = system.encrypt(encoded_plaintext)
        ciphertexts.append((u,c))

    return ciphertexts


def decrypt_ciphertext(ciphertexts: list, system: LWECryptosystem, hamming_decoder: np.arra
    n = system.n
    segment_bits = n//2 + 1
    #each character has 7-bit ascii code
    segment_chars = segment_bits // 7

    decrypted_message = ''
    for text in ciphertexts:
        u, c = text
        decrypted_text = system.decrypt(u, c)

        decoded_text = np.matmul(decrypted_text, hamming_decoder) % 2
        decoded_sum = sum(decoded_text)
        if decoded_sum == segment_bits-1:
            #first row of hamming_decoder is all 1s, so there was an error in the first bi
            decrypted_text[0] = (decrypted_text[0] + 1) % 2
        if sum(decoded_text) == segment_bits-2:
            #one of the next (segment_bits-1) message bits
            #error in the bit of wherever the 0 is plus 1
            error_location = np.argmin(decoded_text) + 1
            decrypted_text[error_location] = (decrypted_text[error_location] + 1) % 2
        #otherwise there's no error or error is in added hammond bits, so we don't care
        corrected_plaintext = decrypted_text[:segment_bits]

        binary_message = ''
        for bit in corrected_plaintext:
            binary_message += str(bit)

        #strip off bits 7 at a time and convert back from ascii to characters
        for i in range(0, len(binary_message), 7):
            decrypted_message += chr(int(binary_message[i:i+7], 2))

    return decrypted_message


def main():
    # characters per message segment
    segment_chars = 5
    # bits per message segment (each char encoded as 7-bit ascii)
    segment_bits = segment_chars * 7
    n = segment_bits * 2 - 1
    l = n
    m = 2008
    q = 2003
    r = 1
    t = 2
    alpha = 0.0065
    system = LWECryptosystem(n, l, m, q, r, t, alpha)

    #rounding procedure occasionally produces an error, so use a hamming code
    hamming_M = np.ones([1,segment_bits-1])
    for i in range(segment_bits-1):
        row = np.ones([1,segment_bits-1])
        row[0,i] = 0
        hamming_M = np.append(hamming_M, row, axis=0)

    hamming_encoder = np.append(np.identity(segment_bits), hamming_M, axis=1)
    hamming_decoder = np.append(hamming_M, np.identity(segment_bits-1), axis=0)

    message = 'learning_with_errors_cryptosystem'
    print(message)
    ciphertexts = send_ciphertext(message, system, hamming_encoder)
    # print(ciphertexts)
    received_message = decrypt_ciphertext(ciphertexts, system, hamming_decoder)
    print(received_message)
```

```
if __name__ == '__main__':
    main()
```