Lattice based cryptography and Learning with Errors

October 17, 2024

Contents

Overview

In these notes, we're going to be introduced to lattice-based cryptography, specifically based on the LWE problem. We'll start by defining a lattice. Next, we'll go onto why lattices are sensible objects to start thinking about for cryptography (and we'll see how they are linked to a hard problem). We then switch to the LWE problem (dipping our toes into what this looks like in the ring and module variants, and the shortest vector problem that is related to its hardness). Next, we discuss a (fairly inefficient) cryptosystem based on LWE. Finally (although we didn't cover this in lectures), we look at FrodoKEM (a more practical key encapsulation mechanism) that is based on the same ideas. Although these notes are intended to be read in order, they don't actually have to be. You can jump around the sections that interest you most.

1 What is a lattice

A lattice in two dimensions includes probably exactly what you imagine: a Cartesian product of points, in a grid structure. In n-dimensions, a lattice is a set of points in n-dimensional space with a

periodic structure. Mathematically a lattice is a n-dimensional vector space over $\mathbb{Z}$, so it is defined by a basis $b_1, \ldots, b_n \in \mathbb{R}^n$; the lattice is given by

$$\mathcal{L}(b_1, \ldots, b_n) := \left\{ \sum_{i=1}^{n} a_i b_i : a_i \in \mathbb{Z} \right\}.$$

A somewhat boring example of a lattice is $\mathbb{Z}^n$.

A lattice is defined entirely by the basis - these are (at most) n *linearly independent* lattice points/vectors/elements (those three are equivalent, so pick the word you like best). The basis is not unique, although all bases of a lattice are equivalent (in the sense that they generate the same lattice, and only that one). Then, given the basis, we can add integer combinations of the n basis elements together to generate all other lattice points. Importantly, we're getting some periodic structure (i.e. when I look at a different section of my lattice, it still looks the same as before), and this structure is discrete (i.e. there's always a gap between any two lattice points, whereas for example if I take the real plane, then I can find two points arbitrarily close together).

A more interesting example of a lattice in two dimensions is the lattice $\mathcal{L}$ defined by

$$\mathbf{u} = \begin{pmatrix} 2 \\ 3 \end{pmatrix} \text{ and } \mathbf{v} = \begin{pmatrix} 1 \\ 4 \end{pmatrix}.$$

These lattices are linearly independent because there do not exist non-zero integers $a, b \in \mathbb{Z}$ so that $a\mathbf{u} + b\mathbf{v} = (0, 0)$. Elements in this lattice are of the form $a\mathbf{u} + b\mathbf{v}$, so for example the points

$$\begin{pmatrix} 4 \\ 6 \end{pmatrix}, \begin{pmatrix} 3 \\ 7 \end{pmatrix}, \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \begin{pmatrix} 0 \\ -2 \end{pmatrix}, \begin{pmatrix} 5 \\ 0 \end{pmatrix}$$

are in $\mathcal{L}$. On the other hand, the point $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$ cannot be written as an integer-linear combination of $\mathbf{u}$ and $\mathbf{v}$ so is not in the lattice.

2 One-way functions from lattices

The point of this section is to show that there is some 'hard problem' related to lattices. This is important to cryptography[a] because otherwise decryption (without knowing the key) would be easy. This section is about *hash functions* - they're a core cryptographic primitive, although a different kettle of fish from encryption/decryption. The fact that there is some hard problem associated with lattices means that there's hope for encryption/encapsulation.

[a] For a deep-dive into this, here's a great paper (that is not very related to this lecture): https://stuff.mit.edu/afs/sipb/project/reading-group/past-readings/2009-06-08-five-worlds.pdf

For lattices to be a candidate for cryptography, we must be able to associate them to a one-way function. A one-way function is a function that is easy to calculate, but computing the input of a

given output is hard. For example, classically we think that multiplication is easy (because I can multiply to n bit numbers together in poly(n) time), but we hope that factoring (a product of two large primes) is hard – this is the hope that underlies RSA. (Of course we've seen that quantumly, this dream is broken; classically it's still not on the strongest footing: the Number Field Sieve is the current best approach, which is a sub-exponential time algorithm.) Ajtai[1] introduced a lattice-based one-way function assuming the hardness of the Shortest Vector Problem, specifically the n^c-approximate Shortest Vector Problem.

The Shortest Vector Problem (SVP) is to find the shortest vector for a given lattice. In the α-approximate SVP, it is enough to find a vector whose length is at most a factor of α greater than the length of the shortest vector.

The SVP is NP-hard. Let's unpack this: a problem is in NP if it can be verified by a non-deterministic Turing machine in polynomial time (in words: it's quick to check if the an instance of the problem is true or false; typically, we assume that instance of the problem is *not* quick to solve). A problem is NP-hard if it's at least as difficult to solve as a problem in NP (in words: in a best-case scenario, it's quick to check, but actually it doesn't need to be). Briefly, we think NP-hard is very hard. The subset-sum problem is perhaps a more familiar NP-hard problem.

The SVP is hard in terms of the dimension of the lattice. So in two dimensions, this problem is not hard, but as the number of dimensions grows, the time/memory required to solve the problem grows.

Ajtai's one-way function is over $\mathbb{Z}_q$; it's actually a hash function.

Very loosely, in a hash function: (i) collisions must be rare (i.e. it's hard to find x, y so that $h(x) = h(y)$); (ii) it's hard to invert (i.e. given an output y, it's hard to find x so that $h(x) = y$; (iii) it's hard to 'forge' outputs: given x, y so that $h(x) = y$, it's hard to find x' so that $h(x') = y$.
These properties are known as (i) collision resistance (ii) pre-image resistance and (iii) second-preimage resistance.

Ajtai's one-way function is a family of functions, parameterised by a matrix A:

$$f_A : \{0, \dots, d-1\}^m \to \mathbb{Z}_q^n$$
$$y \mapsto Ay \pmod q$$

A is a matrix chosen uniformly randomly from $\mathbb{Z}_q^{n \times m}$. For example, a common choice of parameters is $d = 2$, $q = n^2$ and $m > n \log(n^2)$. Because $d = 2$, the input is an m-bit binary string.

Let's have a brief look as to why this is a hash function, by looking at the collision resistance part:
Suppose we have a collision (which would violate the hash function assumption); then there are inputs y, y' so that $Ay = Ay'$. Now let's define an auxiliary, related, lattice[2] $\mathcal{L}' := \{y \in \mathbb{Z}^m : Ay = 0$

[1] M. Ajtai. Generating Hard Instances of Lattice Problems (1996). https://dl.acm.org/doi/pdf/10.1145/237814.237838

[2] Exercise: show that $\mathcal{L}'$ is a lattice.

mod $q\}$. $y-y'$ is a lattice point of $\mathcal{L}'$: that's because $A(y-y') = Ay - Ay'$, which is zero (in $\mathbb{Z}_q^n$) by our assumption of a collision. Now, $y-y'$ is a short vector in the lattice $\mathcal{L}'$: this is because y and y' are made up of only elements from $\{0,\ldots,d-1\}$ (so that $y-y' \in \{-(d-1),\ldots,2d-2\}^m$), whereas lattice points in $\mathcal{L}'$ can have elements being *any* integer: this makes d 'small' and consequently $y-y'$ 'short'. This means that by finding a hash collision, we've just solved the shortest vector problem (or at least the approximate version)! The latter is hard to do, so our assumption of a collision must also be hard.

Notice that the function f_A is simple to implement (because it's just matrix multiplication and modular arithmetic), but the 'key size' (the size of A) grows at least quadratically. We could make this construction more efficient by giving A a special structure.

3 LWE

In this section, we'll learn about the LWE problem: this is how we'll build cryptography from lattices.

3.1 Learning from parity with errors

For $n \geq 1$ and $\epsilon \geq 0$ the *learning from parity with error* problem is to find $\mathbf{s} \in \mathbb{Z}_2^n$ given a list of 'equations with errors':

$$\begin{aligned} \mathbf{s}\cdot\mathbf{a}_1 &\approx b_1 + e_1 \pmod 2 \\ \mathbf{s}\cdot\mathbf{a}_2 &\approx b_2 + e_2 \pmod 2 \\ &\vdots \end{aligned}$$

where $\mathbf{a}_i \in \mathbb{Z}_2^n$ is a vector that is chosen uniformly and independently, and $\cdot$ is the dot product. On the right hand side, b_i is the 'true' answer to the equation $\mathbf{s}\cdot\mathbf{a}_i = b_i$ and we have $b_i \in \mathbb{Z}_2$. However, each of our 'equations' is given to us with a potential error added to it: we see an error with probability $\epsilon \in [0,1]$, and denote this error by e_i. Because of the presence of errors, we can't write out equations with a true equals sign, so we'll use $\approx$ to mean that the equations are approximately equal.

Q. How hard is this problem when $\epsilon = 0$? What value of ϵ makes this problem the hardest?

Q. Imagine that we want to just recover the first bit of s: we could pick a random set of n equations. What probability do we have that our guess for the first bit of s is correct? From this probability, how many times do we need to iterate to be confident of the first bit of s? How many times do we need to iterate to get all of s with a high confidence?

3.2 Learning with errors

We can extend the Learning from Parity with error problem in many ways: firstly, why limit ourselves to working mod 2? Instead, let's work mod p. Secondly, instead of our errors being chosen uniformly, let's choose errors from a probability distribution χ.

This gives us the *learning with error problem*: for $n \geq 1$ and $p = p(n) \leq poly(n)$ prime, find $s \in \mathbb{Z}_p^n$ given

$$\begin{aligned} \mathbf{s}\cdot\mathbf{a}_1 &\approx b_1 + e_1 \pmod 2 \\ \mathbf{s}\cdot\mathbf{a}_2 &\approx b_2 + e_2 \pmod 2 \\ &\vdots \end{aligned}$$

where $a_i \in \mathbb{Z}_p^n$ are chosen uniformly and independently and $e_i \in \mathbb{Z}_p$ is chosen independently according to χ. We refer to this as $\text{LWE}_{p,\chi}$

Q. As a quick exercise to become more familiar with the notation, how would we write the Learning from Parity with Noise problem using the notation $\text{LWE}_{p,\chi}$?

In the LWE problem, we start with an easy problem of "here's a set of more than n equations with n unknowns (the secret); find the unknowns". This problem is easy because we can solve it using linear algebra (i.e. rewrite the equation as a matrix multiplied by a vector of unknowns). One way to see why this is lattice-based cryptography is because we can now reinterpret the matrix as containing basis vecators for our lattice (assuming some details like that our matrix is full-rank). Actually the best reason to see why it's related to lattice-cryptography is because it's lattice methods that are best at cryptanalysis.

To make our problem hard, we obscure our nice linear algebra equation by adding errors. Technically, we're adding errors to *all* equations. However, we need to choose our errors to be small (so that we don't obscure our equation so much that it's unsolvable), which means that a lot of the errors will be zero – i.e. no error. In practice, we're using numbers like $q = 2^{16}$, and the errors are all roughly between 10 and $-10 \pmod q$.

The **Decision LWE Problem** is as follows: given m independent samples $(\mathbf{a}_i, b_i) \in \mathbb{Z}_p^n \times \mathbb{Z}_p$, where either all the samples are of the form $b_i = \mathbf{a}_i \cdot \mathbf{s} + e_i$ (where s is the LWE secret, and e_i the error distributed according to χ), or all the samples are of the from $b_i = u_i$ for some uniformly sampled $u_i \in \mathbb{Z}_p$, decide (with non-negligible advantage) which case we are in.

The **Search-LWE problem** is as follows: given m independent samples $(\mathbf{a}_i, b_i) \in \mathbb{Z}_p^n \times \mathbb{Z}_p$, where $b_i = \mathbf{a}_i \cdot \mathbf{s} + e_i$ (where s is the LWE secret, and e_i the error distributed according to χ), recover $\mathbf{s}$.

Some observations:

- Note that without the error term, both the search and decision problems are easy.
- We can also work with a single matrix equation $(A, As + e)$ instead of considering m samples
- LWE is somewhat similar to code-based cryptography: this time, instead of recovering a codeword, we're recovering a lattice point.
- The decision and search LWE problems are equivalent (up to a polynomial difference in the size of m between the two problems)
- Although I've been writing p a prime, actually it can be a prime power (where typically the notation is $q = p^r$); typically q is a power of 2.

3.3 Ring and Module Learning with errors

We can extend this to Ring or Module LWE by working over a ring or a module instead.

A ring is a group under addition that has a 'multiplication' operation. However, unlike a field, there is not necessarily inverses for non-zero elements. For example $\mathbb{Z}[x]$, the polynomial ring over the integers, is a ring.

A module is a generalisation of a vector space. However, the field of scalars (often $\mathbb{C}$) is replaced by a ring. For example, Consider the module $\mathcal{M}$ with basis elements given by $e_1 = (x, 0), e_2 = (0, x+1)$ and scalar multiplication over $\mathbb{Z}[x]$. Then any element of $\mathcal{M}$ looks like $a_1e_1 + a_2e_2$ where $a_1, a_2 = \mathbb{Z}[x]$.

Kyber, which has been standardised in the form of ML-KEM by NIST, is based on the *module*-LWE problem. This makes it more efficient. On the other hand, the algebraic structure of the module (compared with say FrodoKEM, which has no structure) means that there's potentially something to exploit here. Luckily, we don't yet know how to.

3.4 SVP

The shortest vector problem is essentially the core hard problem underlying LWE. There's actually a couple of different reductions that reduce the hardness of LWE to very similar problems.

The shortest vector problem (SVP) is to find the shortest vector in the lattice. We already saw this for the hash-function earlier.

This is related to the approximate-SVP problem: find a short vector that is at most a (given) factor larger than the shortest vector.

Similarly, we have the shortest independent vectors problem (SIVP): find a collection of independent vectors that form a basis of the lattice that are as short as possible (in the sense that if you line up the vectors from smallest to largest, the largest vector has a minimal size over all possible sets of independent vectors). This is related to the approximate-SIVP problem, in which you're allowed to be a given factor away from the shortest possible independent set.

The hardness of lattice based cryptography is based on problems of this flavour.

4 Cryptosystem

There are a number of different ways that we can encrypt using LWE. They're all somewhat similar, and the differences are usually to take advantages of various efficiencies.

4.1 Encrypting and decrypting

In this subsection, we'll describe a system that encrypts a message. The goal is for decryption to recover that same message.

Our cryptosystem is parameterised by integers n (the security parameter), m (number of equations), q (modulus), and χ (noise distribution).

- The private key is $\mathbf{s} \in \mathbb{Z}_q^n$, chosen uniformly
- The public key is $(A, \mathbf{t} = A\mathbf{s} + \mathbf{e}) \in \mathbb{Z}_q^{m \times n} \times \mathbb{Z}_q^m$, where $\mathbf{e} \in \mathbb{Z}^m$ is the error vector chosen so that each element is independently chosen randomly according to the probability distribution χ.

- We will encrypt a message in $\{0,1\}^*$. For each bit of the message, chose a random set S uniformly among all 2^m subsets of $[m]$. The encryption is $(\sum_{i\in S}\mathbf{a}_i, \sum_{i\in S} t_i)$ if the message bit is 0, and $(\sum_{i\in S}\mathbf{a}_i, \lfloor\frac{q}{2}\rfloor + \sum_{i\in S} t_i)$ if the message bit is 1. Here, $\mathbf{a}_i$ is the ith row of the matrix A, and t_i is the ith element of the vector $\mathbf{t}$.
- To decrypt a pair $(\mathbf{a}, b)$, output 0 if $b - \langle\mathbf{a},\mathbf{s}\rangle$ is closer to 0 than to $\lfloor\frac{q}{2}\rfloor$ (modulo q) and 1 otherwise.

Note that this system is quite inefficient.

Let's do an example: We'll set: $n = 3$, $m = 5$, $q = 8$.
Our errors will be 0 with probability $\frac{1}{2}$, ±1 with probability $\frac{1}{5}$ and ±2 with probability $\frac{1}{20}$. (Note that this is a probability distribution because $\frac{1}{2} + 2\times\frac{1}{5} + 2\times\frac{1}{20} = 1$.)

We'll choose our secret vector $\mathbf{s}$ uniformly randomly in $\mathbb{Z}/q\mathbb{Z}$: $s = (1,7,3)$.

Similarly, we'll choose our matrix A so that each element is chosen uniformly randomly from $\mathbb{Z}/q\mathbb{Z}$:

$$A = \begin{pmatrix} 0 & 7 & 3 \\ 5 & 4 & 3 \\ 5 & 2 & 7 \\ 3 & 0 & 7 \\ 3 & 4 & 1 \end{pmatrix}$$

Finally we sample $\mathbf{e}$ according to our distribution so that $\mathbf{e} = (0,0,0,1,-1)$.

This means that $\mathbf{t} = A\mathbf{s} + \mathbf{e} = (2,2,0,1,1)$.

Now let's encrypt a message $m = 0$ using $S = \{0,2\}$ (i.e. we'll take the first and third rows of A). We can then calculate the ciphertext as:

$$c_0 = ((5,1,2),2)\ .$$

Similarly, if we had used the same set S to encrypt the message $m = 1$, we would have the ciphertext

$$c_1 = ((5,1,2),6)\ .$$

Now, let's decrypt: using our ciphertext c_0, we recover the element 0: this is clearly closer to 0 than to $q/2 = 4$, so we decrypt to get the message 0.
Using our ciphertext c_1, we recover the element 4: this is clearly closer to $q/2 =$ than to 0, so we decrypt to get the message 1.

Question: how big is the public key? By how much does the ciphertext increase for each message bit that we send?

Exercise: have a go at coding this encryption/decryption (Sagemath[3] is a good choice if you're otherwise ambivalent as to language.) I've included my version at the end of these notes.

[3] https://sagecell.sagemath.org/

Correctness A major question that we need to ask, is whether this cryptosystem work, in the sense that decryption recovers the message.

Exercise: show that decryption works if there is no error.

For S associated to a given message bit, decryption works as long as $\sum_{i \in S} e_i \leq q/4$. Therefore, the probability of a decryption failure is the probability that this inequality is *not* satisfied. The probability of a decryption failure therefore depends on the error distribution; typically we will choose errors following a discrete Gaussian distribution and we use standard probability bounds (or even just direct calculation, depending on the size of q) to bound this probability. We want the failure probability to be small.

Security The other major question with which we must concern ourselves is whether this cryptosystem is secure, in the sense that the public data (public key and ciphertext) does not reveal anything about the message. In reality, we will be satisfied if only a negligible amount of information about the message is revealed.

We will sketch the security proof against chosen plaintext attacks.

Suppose that there exists an efficient algorithm that, given a public key $(A, \mathbf{t})$ as above, can correctly guess the encrypted bit with probability at least $\frac{1}{2} + \frac{1}{poly(n)}$. Now, let us input into the algorithm $(A, \mathbf{u})$, where $\mathbf{u} \in \mathbb{Z}_q^m$ is chosen uniformly, and a random bit encrypted using $(A, \mathbf{u})$. Then, it follows [4] that with very high probability that the distribution $(\sum_{i \in S} \mathbf{a}_i, \sum_{i \in S} t_i)$ is close to uniform. Consequently, encryptions of 0 and 1 are pretty much identically distributed and so the algorithm cannot distinguish the encrypted bit beyond random guessing. So either the algorithm *can* correctly guess the value of an encrypted bit (if it's given a 'proper' sample) or it cannot (if it's given a random sample); this means it can distinguish LWE samples from uniform samples. This violates the decision LWE problem.

4.2 Key Encapsulation - FrodoKEM

Encapsulation is very related to encryption: we want two (or more) parties to agree a random secret. For example, Alice could send to Bob an encrypted secret that she has chosen. However, when encapsulating in general, Alice does not necessarily have control over the randomness generated. As typical encryption involves first using public key cryptography to exchange a key, and then symmetric cryptography.

The Key Encapsulation Mechanism (KEM) that we describe is basically FrodoKEM. It is defined over $\mathbb{Z}_q$, where $q = 2^N$ is a power of 2.

> FrodoKEM is heavy in notation: in particular, the rounding and cross-rounding functions are complicated and confusing. They're introduced to make FrodoKEM more efficient: we get to drop some bits from the elements that we send, so we're sending smaller numbers. The unobvious way that they're defined is a technical detail to make reconciliation (i.e. the step where Alice and Bob work out what key they share) work, and to make sure that everything that's sent looks random.

We'll first introduce some notation: let $r \to \chi^{n \times m}$ to denote that every element of $r \in \mathbb{Z}_q^{n \times m}$ is sampled uniformly at random from $\mathbb{Z}_q$ according to the probability distribution χ.

[4] This claim needs proof. For example, you could use the Leftover Hash Lemma.

Let $B < N - 1$ and $\bar{B} = N - B$. We define the *rounding* function as

$$\lfloor \cdot \rceil_{2^B} : v \mapsto \lfloor 2^{-\bar{B}} v \rceil \pmod 2^B,$$

where $v \in \mathbb{Z}_q$ is represented as an integer in $[0, q)$. The rounding function outputs the B most significant bits of $v + 2^{\bar{B}-1}$, essentially partitioning $\mathbb{Z}_q$ into 2^B intervals.

We will define the *cross-rounding function* from $\mathbb{Z}_q \to \mathbb{Z}_2$ as follows:

$$\langle \cdot \rangle_{2^B} : v \mapsto \lfloor 2^{-\bar{B}+1} v \rfloor \mod 2$$

where $\bar{B} = (\log_2 q) - B$. The cross rounding function partitions $\mathbb{Z}_q$ according to the $(B+1)$th most significant bit.

- Alice generates $A \in \mathbb{Z}_q^{n \times n}$ uniformly at random. She samples $S, E \to \chi^{n \times \bar{n}}$ and sends $(A, AS + E) \in \mathbb{Z}_q^{n \times n} \times \mathbb{Z}_q^{n \times \bar{n}}$.
- Bob samples $S', E' \to \chi^{\bar{n} \times n}$ and $E'' \to \chi^{\bar{n} \times \bar{n}}$. He calculates $B' = S'A + E'$ and $V = S'B + E''$. Bob sends $(B', \langle V \rangle_{2^B} \in \mathbb{Z}_q^{\bar{n} \times n}) \times \mathbb{Z}_2^{\bar{n} \times \bar{n}}$.
- Alice calculates $B'S$; for each element x_{ij} in this matrix, she outputs $\lfloor v \rceil_{2^B}$, where v is the closest element to x_{ij} so that $\langle v \rangle_{2^B} = C_{ij}$. She calls this output matrix K_A.
- Bob calculates $\lfloor V \rceil_{2^B}$, calling this matrix K_B.

Exercise: have a go at coding this, to get an idea of what's going on.

Correctness With high probability, $K_A = K_B$. Hence Alice and Bob agree on a secret. This of course depends on the choice of the error distribution. The error distribution is an approximation to the rounded Gaussian.

Additional Reading

[1] Oded Goldreich, Shafi Goldwasser, and Shai Halevi. "Collision-free hashing from lattice problems". In: *Studies in Complexity and Cryptography* (2011), pp. 30–39. URL: https://www.wisdom.weizmann.ac.il/~oded/COL/cfh.pdf.

[2] Oded Regev. "The learning with errors problem". In: *Invited survey in CCC* 7.30 (2010), p. 11. URL: https://cims.nyu.edu/~regev/papers/lwesurvey.pdf.

[3] FrodoKEM Team. *FrodoKEM: Learning With Errors Key Encapsulation Preliminary Standardization Proposal*. Tech. rep. 2023. URL: https://frodokem.org/files/FrodoKEM-standard_proposal-20230314.pdf.

A Sage Code for encryption

```
n = 3
m = 5
q = 8

# define the error distribution specific to the choice m=5
P5 = [0.05,0.2, 0.5, 0.2,0.05]
chi = GeneralDiscreteDistribution(P5)

# Q let's us sample a random element from Z/qZ
Q = [1 for _ in range(q)]
chiQ = GeneralDiscreteDistribution(Q)

# let's create s, with elements chosen uniformly from Z/qZ
s = []
for _ in range(n):
    ts = chiQ.get_random_element()
    s.append(ts)
s = vector(s)

# let's create A, with elements chosen uniformly from Z/qZ
A = Matrix(IntegerModRing(q), [[0 for _ in range(n)] for _ in range(m)])

for i in range(m):
    for j in range(n):
        a = chiQ.get_random_element()
        A[i, j] += a

# let's create e, with elements chosen independently according to the probability distrubtion chi

e = []
for _ in range(m):
    te = chi.get_random_element()
    e.append(te-2) # this step is to make the error be -2,-1,0,1,2
e = vector(e)

# let's calculate our public key
t = A*s + e

# let's see what we have:
print("s: ", s)
print("A: \n", A)
print("e: ", e)
print("t: ", t)

############ ENCRYPTION ################
S = [0,2] # a choice of S specific to n=3

left = sum(A[i] for i in S) # given S, let's see which equations of A we need

print("Encryption public key wrt S: ", left)

# Encrypt m=0
print("Let's encrypt m = 0")
encrypt0 = sum(t[i] for i in S) # the part of the ciphertext encrypting the message
print("encrypt0: ", encrypt0)

```

```
# Encrypt m=0
print("Let's encrypt m = 1")
encrypt1 = sum(t[i] for i in S) + q//2
print("encrypt1: ", encrypt1)

############ DECRYPTION ################
print("Let's decrypt")
subtract = left.dot_product(vector(s))
decrypt0 = encrypt0 - subtract
decrypt1 = encrypt1 - subtract

# these are the actual values we get on decrypting
print("decrypt 0:" , decrypt0)
print("decrypt 1:" , decrypt1)

# this function will let us interpret the decryption values as m=0 or m=1
def reconcile(dec):
    x = abs(int(dec%q))
    y = abs(int((dec - q//2)%q))
    if x< y:
        return 0
    else:
        return 1


# let's see what we get upon completing encryption
print("reconcile c0: ", reconcile(decrypt0))
print("reconcile c1: ", reconcile(decrypt1))
```