Cryptography Report

Crack 1: Attacking 'Learning Without Errors'

In this version of the LWE cryptosystem, the attacker knows that the error distribution, χ, is constant (at 0), so they can attempt to efficiently solve the underlying equation to extract the private key, **s**. Once the private key is known, they can simply decrypt the ciphertext - using the standard LWE decryption algorithm - with this key to obtain the corresponding plaintext. The public key pair gives (**A**, **b**), so we know that:

$$\mathbf{b} = \mathbf{As} + \mathbf{e}, \quad \mathbf{e} \in \chi^m \quad \Rightarrow \quad \mathbf{b} = \mathbf{As} \quad \Rightarrow \quad \mathbf{s} = \mathbf{Lb}$$

where **L** is the left inverse of **A** (**A** is typically a non-square matrix). Assuming we can compute the left inverse of **A**, we then multiply it with **b** as above to acquire the private key, **s**.

Two conditions must be satisfied for **A** to have a left inverse:

1) The rank of **A** must be equal to the number of columns in **A**, ie. $rank(\mathbf{A}) = n$

2) The number of rows in **A** must be greater than the number of columns, ie. $m > n$

The second condition is typically always true, since more LWE equation instances (m) are given than the dimension of the private key (n), in an attempt to mitigate the number of incorrectly decrypted bits. The first condition has a very high likelihood of being true as well, since **A** usually consists of n random vectors from $\mathbb{Z}_q^m$, where m is considerably larger than n. Since these column vectors are random, it is very likely that they are linearly independent - ie. there is a leading one in each column of the row-reduced echelon form, after applying Gauss-Jordan elimination. Conversely, for m considerably larger than n, it would be very unlikely for there to be at *least* m-n rows which become zero rows under Gauss-Jordan elimination from less than n rows. Therefore, we can assume the left inverse, **L**, exists.

L can be computed as so:

$$\mathbf{L} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T$$

Two conditions must be satisfied in order for $(\mathbf{A}^T\mathbf{A})^{-1}$ to exist:

1) The determinant of $\mathbf{A}^T\mathbf{A}$ must be non-zero

2) Every element in the matrix $\mathbf{A}^T\mathbf{A}$ must have a multiplicative inverse modulo q

The first condition trivially holds from the previous result confirming that the rank of **A** is very likely to be n. The second condition will also be true if every element in $\mathbf{A}^T\mathbf{A}$ is coprime to q (ie. $\forall\ x \in \mathbf{A}^T\mathbf{A} : \gcd(x, q) = 1$). Since q is set to be a prime number, this will hold - no number smaller than a prime number shares any other factors with said prime number except for 1. This is a famously known result whereby $\mathbb{Z}_p$ forms a field, rather than a ring, when p is prime, thus allowing for multiplicative inverses to exist.

Although this procedure for crack 1 may not always be successful, it is very likely to work, and only utilises simple and computationally efficient - both in time and memory - matrix operations to extract **s** from the public key. This is especially useful when considering the relatively large parameters this algorithm will be tested on (up to $n = 256$, $m = 2048$, and $q = 10007$). The galois package was used to make computing inverse matrices modulo q efficient, without having to create a separate subroutine, since the standard *NumPy* method did not allow for efficient matrix inversing in modular arithmetic.

Crack 2: Attacking 'Learning With a Few Errors'

The basis of this attack is a brute force search on the error vector $\mathbf{e} \in \chi^m$, exploiting the fact that most of the elements in **e** are expected to be 0, with a small number of elements being either 1 or -1. As such, the search parameter r is initially

set to be 0, increasing by 1 after all of the options at this iteration of *r* have been exhausted. *r* represents the number of elements in **e** which are non-zero, which is why starting the search in an ascending fashion from 0 is a good idea, since it is more likely that **e** has fewer non-zero elements. At each iteration of *r*, we build a list called *ind_combs*, storing all of the index combinations of **e**, ie. mCr, using the standard library *itertools*. We also create *it_opt*, which is the cartesian product of the sets {1, -1}, {1, -1}... repeated *r* times. This variable stores all of the different error element permutations and is used repeatedly for each index combination specified by *ind_combs*. Essentially, *ind_combs* is concerned with all of the possible locations in **e** of the non-zero elements, and *it_opt* provides every permutation of the non-zero options to try at each index combination in the current iteration of *r*. We build each of the possible error vectors in this way, and run the trial code, terminating if we find a match or if we have searched all of the possible cases. Before explaining the trial code and what constitutes as a match, here is an example of how **e** is generated with this method, (m = 5):

r	*ind_combs*	*it_opt*	**e**
2	(0, 1), (0, 2), (0, 3), (0, 4), (1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)	(1, 1), (1, -1), (-1, 1), (-1, -1)	[1, 1, 0, 0, 0], [1, -1, 0, 0, 0], [-1, 1, 0, 0, 0], [-1, -1, 0, 0, 0], [1, 0, 1, 0, 0],[1, 0, -1, 0, 0], [-1, 0, 1, 0, 0], [-1, 0, -1, 0, 0], [1, 0, 0, 1, 0], [1, 0, 0, -1, 0], [-1, 0, 0, 1, 0], [-1, 0, 0, -1, 0], [1, 0, 0, 0, 1], [1, 0, 0, 0, -1], [-1, 0, 0, 0, 1], [-1, 0, 0, 0, -1], [0, 1, 1, 0, 0], [0, 1, -1, 0, 0], [0, -1, 1, 0, 0], [0, -1, -1, 0, 0] ...

The trial code is as follows. We know:

$$\mathbf{b} = \mathbf{As} + \mathbf{e}, \quad \mathbf{e} \in \chi^m \quad \Rightarrow \quad \mathbf{b} - \mathbf{e} = \mathbf{As} \quad \Rightarrow \quad \mathbf{L}(\mathbf{b} - \mathbf{e}) = \mathbf{s}$$

where **L** is the left inverse of **A**. Assuming we guess the correct **e**, this gives us the correct private key **s**. Now looking at the basis of the lattice formed from the public key (**A**, **b**) - as seen in the lectures, we have:

$$\begin{vmatrix} \mathbf{A} & \mathbf{b} \\ 0 & 1 \end{vmatrix} \cdot \begin{vmatrix} \mathbf{x} \\ y \end{vmatrix} = \begin{vmatrix} \mathbf{Ax} + \mathbf{b}y \\ y \end{vmatrix}$$

where the vector $|\mathbf{x}\ y|^T$ represents the arbitrary integer coefficients of each basis vector, resulting in another point in the lattice space given as $|\mathbf{Ax} + \mathbf{b}y \quad y|^T$. Using the theorem that the shortest vector in this lattice is typically $|\mathbf{-e} \quad -1|^T$, (especially now that **e** is expected to be even shorter, since there are fewer non-zero elements than the **e** in standard LWE encryption), we obtain:

$$\begin{vmatrix} \mathbf{A} & \mathbf{b} \\ 0 & 1 \end{vmatrix} \cdot \begin{vmatrix} \mathbf{s} \\ -1 \end{vmatrix} = \begin{vmatrix} \mathbf{As} - \mathbf{b} \\ -1 \end{vmatrix} = \begin{vmatrix} \mathbf{-e} \\ -1 \end{vmatrix}, \quad \text{where: } \mathbf{As} - \mathbf{b} = \mathbf{-e} \text{ (from } \mathbf{b} = \mathbf{As} + \mathbf{e})$$

If we use the **s** we extracted earlier in our guess of **e**, we would expect the above matrix multiplication to give us back -1 times the **e** we brute-forced with to begin with. In essence, for every guess of **e**, we are checking if: $\mathbf{A}(\mathbf{L}(\mathbf{b} - \mathbf{e})) - \mathbf{b} = \mathbf{-e}$, where **L** can be computed as explained in crack 1. If the left hand side of this equation matches the negative of the **e** we are brute forcing with - at this trial - for every element, we can: conclude that we have guessed the correct **e**, terminate the search and extract $\mathbf{L}(\mathbf{b} - \mathbf{e})$ as **s**, the private key. All we have to do now is decrypt the given ciphertext with this key to obtain the correct plaintext.

Although the search space for this brute force algorithm is very large (the number of possible **e**'s grows exponentially), we expect it to terminate early into the search because of the fact that there are only a few errors and we search from r = 0 to m. The trial code itself uses simple matrix calculations, as in crack 1, making this method more computationally efficient than using typical enumeration/sieving strategies to solve it as an SVP, especially since the parameters are considerably larger than those in crack 3.

Crack 3: Attacking 'Learning With Errors'

The approach used for this attack is to create the lattice basis from the public key (**A**, **b**) as in crack 2:

$$\begin{vmatrix} \mathbf{A} & \mathbf{b} \\ 0 & 1 \end{vmatrix} \cdot \begin{vmatrix} \mathbf{x} \\ y \end{vmatrix} = \begin{vmatrix} \mathbf{Ax} + \mathbf{b}y \\ y \end{vmatrix}$$

and formulate the problem as an SVP, whereby the shortest vector in this lattice is very likely to be that of the error vector **e**. As seen previously in crack 1 & 2, once the error vector is found (the shortest vector), the underlying LWE equation can be solved, using the left inverse of A, to obtain the private key **s**. The ciphertext can then be decrypted as usual with **s**.

The method used for solving this SVP is a difference sieve, since these algorithms are quicker than enumeration methods on average, despite typically using more memory. The first step is to create a bag of vectors from random linear combinations of the basis vectors. The main loop of the procedure will continue while there exists two different vectors in the bag, **v**, **w**, such that: $||\mathbf{v} - \mathbf{w}|| < ||\mathbf{v}||$. If this is the case, we update the vector **v** to be **v** - **w**, and the loop proceeds, updating the index in our bag containing the shortest vector we have managed to find along the way. Once the loop terminates, we check that we have found the shortest vector - typically only requiring a check that y is 1 or -1. If this is not the case, we recreate the bag of vectors and repeat the procedure with this new random bag until we find the shortest vector. In testing, we found that a *bag_size* of 200 will find the shortest vectors in one loop (ie. no repeats) for the given parameters (n = 5, m = 48, and q = 19) roughly 90% of the time, running for around 1m to 1m 20s on the testing device.

It is important to discuss the normalisation function used in comparing vector sizes. We use the standard L2 (Euclidean) norm, but larger elements may use a smaller value in the L2 formula to accommodate for the modular arithmetic. We build a list called *norm_aux*, which stores the values that should be used when calculating norms, so that an element of 18 and 1 are considered equally close in magnitude to 0 when working in modulo 19, for example (as with q - 2 and 2, q - 3 and 3, etc..). Below is an example calculation of the norm of a vector, modulo 19:

$$norm\_aux_{19} = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, -9, -8, -7, -6, -5, -4, -3, -2, -1]$$

$$\mathbf{v} = |\, 2 \;\; 0 \;\; 18 \;\; 10 \;\; 4 \,|^{T}, \qquad ||\mathbf{v}|| = \sqrt{\sum_{i \; in \; v} norm\_aux[i]^2} = (2^2 + 0^2 + (-1)^2 + (-9)^2 + 4^2)^{0.5} \simeq 10.0995$$

Unlike other sieve algorithms, the number of vectors in the bag in this algorithm doesn't increase exponentially (it remains constant), resulting in a more time and memory efficient algorithm which exploits the smaller parameter space used. Furthermore, using averages between vectors to compute new shorter vectors requires amendments to ensure that valid lattice vectors are produced (eg. altering the coordinates by 1 or -1), and this extra computation can slow down the procedure when many iterations are performed. This is another reason why the difference method is preferred, since it is easier to compute and guarantees a valid lattice vector every time. The main drawback is that it is possible to repeat the procedure entirely if the bag of vectors converges too quickly into a local minimum of vector sizes; however, by manipulating the *bag_size* to be large enough, we can mostly ensure a gradual decrease in vector sizes until the global minimum (shortest vector) is found, whilst maintaining a reasonably small bag of vectors for efficiency. The LLL algorithm to improve the orthogonality of the basis vectors is not used, since n random vectors in a much higher dimension, m, typically have large enough angles of deviation in the lattice space to generate useful vectors in our search. Therefore, the computational cost of including this algorithm before the main loop is not worth the slight improvement, if any, at runtime for this sieving method.