

ECDLP

(Elliptic Curve Discrete Logarithm Problem)

Sanjana Joshi

MSc(Scientific Computing), Department of Scientific Computing, Modeling and Simulation
Savitribai Phule Pune University

May 10, 2025

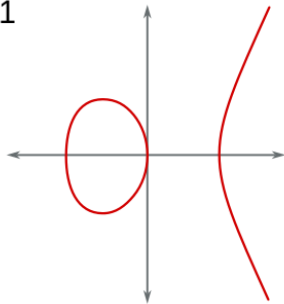
Things that this presentation covers

1. What exactly is ECDLP?
2. Why calculating 'k' is impossible as of today?
3. Applications of ECDLP
4. What is the Zero Minor Problem?
5. Scale of the problem
6. Project aim
7. Approach for solving the problem
8. Implementation done until now
9. How large a matrix has the program been tried out on?
10. Cluster setup
11. Things I have learned till date

Elliptic Curve

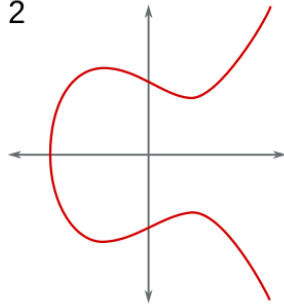
$$\text{Equation : } y^2 = x^3 + ax + b$$

1



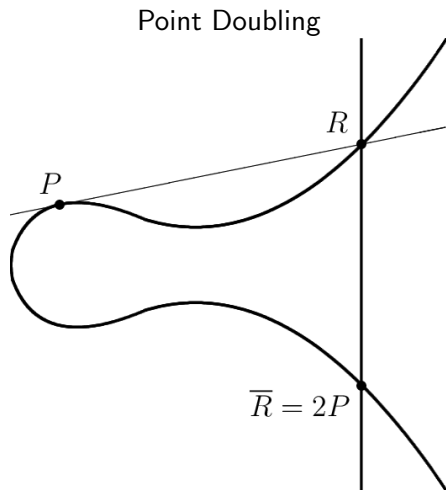
$$y^2 = x^3 - x$$

2

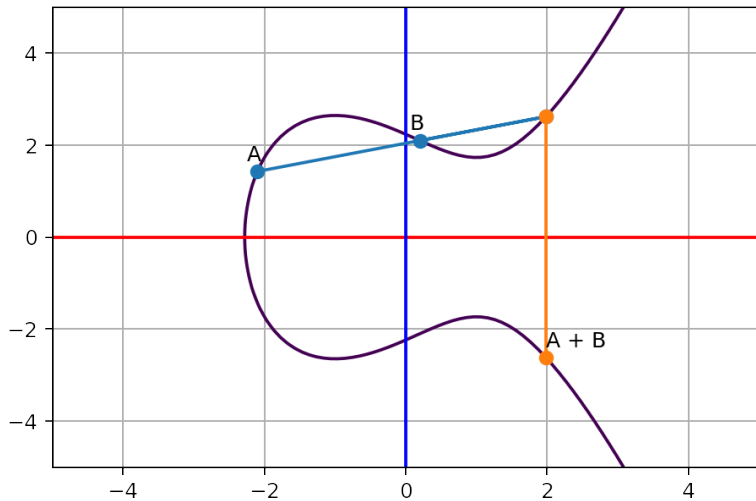


$$y^2 = x^3 - x + 1$$

Point Arithmetic on an Elliptic Curve



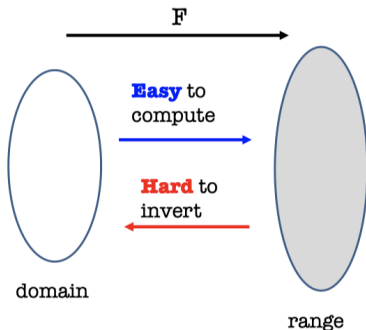
Point Adding



The Elliptic Curve Discrete Logarithm Problem

In case of Elliptic Curve Cryptography (ECC), P and Q are points present on an Elliptic Curve, and the Elliptic Curve Discrete Logarithm Problem is finding out an integer ' k ' such that $Q = k \cdot P$

Security of the curve depends on the **order** (the total number of distinct points on the curve) of the curve.



Possible number of private keys

The perceived range of all possible private cryptographic keys is $1.1579209 * 10^{77}$ that is approximately equal to 2^{256}

Private key : k , depends on the **order of the point P**
($'k'$ can be any value from 1 up to 2^{256})

Fun fact : Number of atoms in the observable universe is around 10^{80}

So why is the calculation of this 'k' computationally impossible today?

Let's say a supercomputer can try up to one trillion (10^{12}) values per second.

To go through all 2^{256} values, we would require

$$\frac{1.16 \times 10^{77}}{10^{12}} = 1.16 \times 10^{65} \text{ seconds}$$

Converting that to years gives us

$$\frac{1.16 \times 10^{65}}{60 \times 60 \times 24 \times 365} \approx 3.68 \times 10^{57} \text{ years}$$

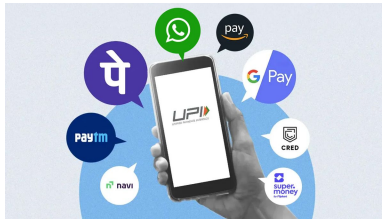
The age of the universe is about 13.8 billion years $\approx 1.38 \times 10^{10}$

So brute force approach for solving ECDLP would take around :

$$\frac{3.68 \times 10^{57}}{1.38 \times 10^{10}} \approx 2.67 \times 10^{47} \text{ universe lifetimes}$$

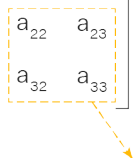
Applications of ECDLP

- **UPI (Unified Payments Interface)** : January 2025 recorded 16.99 billion transactions.
- **Whatsapp** : End-to-End Encryption
- **Blockchain Technology** : Bitcoin and Ethereum (for the creation and verification of digital signatures)
- In general, any place where **SSL (Secure Socket Layer)** comes into picture ECDLP plays an important role. For example, logging into Gmail, Amazon, etc.



Zero Minor Problem

If the matrix received from the LasVegas Algorithm contains a minor with determinant 0, ECDLP can be solved.

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

$$\mathbf{M}_{11} = \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix}$$

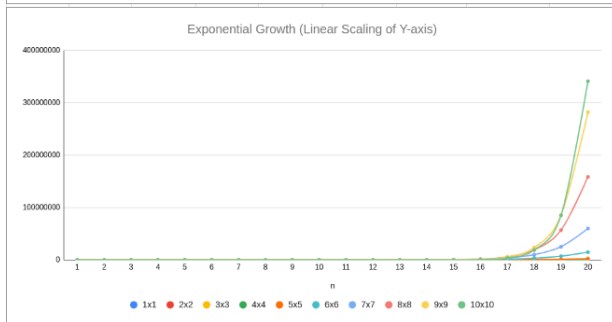
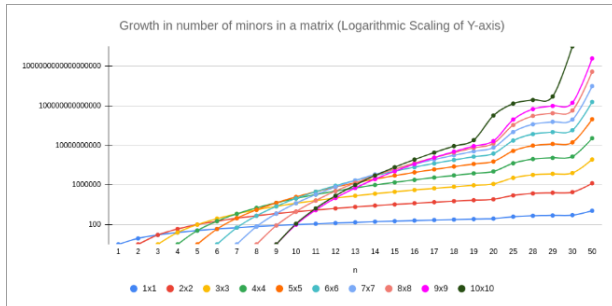
If minor $|\mathbf{M}_{11}| = 0$, ECDLP can be solved.

But, there's a catch

The dimensions of the matrix received from the LasVegas Algorithm are so big, that getting all the possible minors from this matrix will require a very long time which is not possible with the available computational power we have.

So how huge can the number of minors get?

n	1x1	2x2	3x3	4x4	5x5	6x6	7x7	8x8	9x9	10x10
1	1	0	0	0	0	0	0	0	0	0
2	4	1	0	0	0	0	0	0	0	0
3	9	9	1	0	0	0	0	0	0	0
4	16	36	16	1	0	0	0	0	0	0
5	25	100	100	25	1	0	0	0	0	0
6	36	225	400	225	36	1	0	0	0	0
7	49	441	1225	1225	441	49	1	0	0	0
8	64	784	3136	4900	3136	784	64	1	0	0
9	81	1296	7056	15876	15876	7056	1296	81	1	0
10	100	2025	14400	44100	63504	44100	14400	2025	100	1
11	121	3025	27225	108900	213444	213444	108900	27225	3025	121
12	144	4356	48400	245025	627264	853776	627264	245025	48400	4356
13	169	6084	81796	511225	1656369	2944656	2944656	1656369	511225	81796
14	196	8281	132496	1002001	4008004	9018009	11778624	9018009	4008004	1002001
15	225	11025	207025	1863225	9018009	25050025	41409225	41409225	25050025	9018009
16	256	14400	313600	3312400	19079424	64128064	130873600	165636900	130873600	64128064
17	289	18496	462400	5664400	38291344	153165376	378224704	590976100	590976100	378224704
18	324	23409	665856	9363600	73410624	344622096	1012766976	1914762564	2363904400	1914762564
19	361	29241	938961	15023376	135210384	736145424	2538950544	5712638724	8533694884	8533694884
20	400	36100	1299600	23474025	240374016	1502337600	6009350400	15868440900	28210561600	34134779536
25	625	90000	5290000	160022500	2822796900	31364410000	231072490000	1169804480625	4173746850625	10684791937600
28	784	142884	10732176	419225625	9658958400	141933027600	1401950721600	9660316691025	47705267610000	172216016072100
29	841	164836	13351716	564110001	14102750025	225644000400	2436034208400	18422508701025	100300325150025	401201300600100
30	900	189225	16483600	751034025	20307960036	352568750625	4144481640000	34256731055625	204694541122500	902702926350225



Given a big matrix, reduced using an anti-offset, the goal is to check whether this reduced matrix can still solve the Zero Minor Problem.

If yes, then :

- 1) What is a good value for the anti-offset (between 0 to 1)?
- 2) Is the anti-offset really useful?

Values of anti-offset tried out till now : 0.2, 0.3

This means shrinking the big matrix by 20-30%

This involves checking if the Zero Minor Problem can be solved using the matrix even if it is shrunk by 50%, then further reducing that 50% shrunk matrix to 25%, and checking again, if the Zero Minor Problem can be solved using this matrix that has been reduced by almost 75%, then we will have a comparatively smaller matrix to deal with, and the chances of solving the Zero Minor Problem will be much higher than attempting to solve ZMP on the original matrix.

Approaches for solving the problem

- Approach 1 : Brute-Force
- Approach 2 : Checking for Linear Dependence
- Approach 3 : The Latecomer's Algorithm

Approach 1 : Brute-Force (not feasible)

The idea : Getting all the minors of a $n \times n$ matrix starting from 2 to $(n-1)$, and calculating their determinants to check whether any of them is 0.

- Advantages
 - Easy to implement and understand.
- Problems
 - The amount of time required is huge.
 - The available computational power we have is insufficient.

Approach 2 : Checking for linear dependence (no significant results)

The idea : Checking to see if any of the columns have linear dependence, which will make the determinant of some minor 0.

- Problems
 - Searching for linearly dependent rows or columns may involve several row or column searches, and some elements in a row or column may be linearly dependent with some other row or column elements which may make the determinant of the minor 0, but this requires several combinations of rows and columns.

Approach 3 : The Latecomer's Algorithm (still in progress)

The idea : If we calculate the determinants of 2×2 minors, and the corresponding 3×3 minor that contains these 2×2 minors have 2* or more minors with determinant 0, then the 3×3 minor will be worth looking into, as its determinant could be 0, or it may be skipped otherwise. Making use of Probabilistic Pruning, and eliminating the minors whose determinant is less likely to be 0, and considering only those minors whose determinants are very likely to be 0 because of the smaller zero minors that these bigger minors contain. This will reduce the computations of the minors and their determinants up to certain extent. If the algorithm works fine, chain reaction may be seen eliminating the minor calculations even further, for example, a 3×3 minor that has non-zero determinant 2×2 minors will be skipped during calculations, and if the 4×4 minor containing this 3×3 minor has 2 or more skipped adjacent 3×3 minors, then we can skip the calculation of this 4×4 minor determinant as well, and so on.

Implementation done until now (using C++)

- Getting all possible minors from a given $n \times n$ matrix.
 - Using dedicated 'for' loops for each minor present in the matrix (not feasible)
 - Using permutations of array indices
 - Getting all possible combinations serially using matrix indices, and later retrieving the minor using those index combinations (using vectors as well as arrays)
 1. Serial Implementation
 2. Parallel Implementation
- Attempt to implement The Latecomer's Algorithm.

Broadcasting the matrix

```
void broadcastMatrix(mat_ZZ_p &mat, int rank, int n)
{
    if (rank == 0)
    {
        long int flatMat[n * n];
        flattenMatrix(mat, flatMat);
        MPI_Bcast(&n, 1, MPI_LONG, 0, MPI_COMM_WORLD);
        MPI_Bcast(flatMat, n * n, MPI_LONG, 0, MPI_COMM_WORLD);
    }
    else
    {
        long int flatMat[n * n];
        mat.SetDims(n, n);
        MPI_Bcast(&n, 1, MPI_LONG, 0, MPI_COMM_WORLD);
        MPI_Bcast(flatMat, n * n, MPI_LONG, 0, MPI_COMM_WORLD);
        for (long i = 0; i < n; ++i)
        {
            for (long j = 0; j < n; ++j)
            {
                mat[i][j] = conv<ZZ_p>(flatMat[i * n + j]);
            }
        }
    }
}
```

Permutations of indices approach

```
do
{
    vector<int> sortedIndices;
    for (int i = 0; i < MINORSIZE; i++)
    {
        sortedIndices.push_back(arr[i]);
    }

    sort(sortedIndices.begin(), sortedIndices.end());

    if (find(minorIndices.begin(), minorIndices.end(), sortedIndices) == minorIndices.end())
    {
        minorIndices.push_back(sortedIndices);
    }
} while (next_permutation(arr, arr + n));
```

Errors committed in the first parallel implementation

```
long int ind = 0;

long int combinationCnt = nCr(n, r);
long int minorIndRes[r];
int index = (combinationCnt / world_size);

if (world_rank != (world_size - 1))
{
    for (ind = index * world_rank; ind < (index * (world_rank + 1)); ind++)
    {
        get_kth_combination(combinations, n, r, ind, minorIndRes);
        getMinors(minorIndRes, mat, r);
    }
    MPI_Bcast(&ind, 1, MPI_LONG, 0, MPI_COMM_WORLD);
}

MPI_Barrier(MPI_COMM_WORLD);
if ((ind < combinationCnt) && (world_rank == (world_size - 1)))
{
    for (ind = (index * (world_size - 1)); ind < combinationCnt; ind++)
    {
        get_kth_combination(combinations, n, r, ind, minorIndRes);
        getMinors(minorIndRes, mat, r);
    }
}
```

Serial implementation

```
void getMinors(long int combinations[], long int combinationCount, int n, int r)
{
    long int col[r], row[r];
    long int minorCount = 0;

    for (int combinationNumber1 = 0; combinationNumber1 < combinationCount; combinationNumber1++)
    {
        get_kth_combination(combinations, n, r, combinationNumber1, row);
        for (int combinationNumber2 = 0; combinationNumber2 < combinationCount; combinationNumber2++)
        {
            get_kth_combination(combinations, n, r, combinationNumber2, col);
            for (int i = 0; i < r; i++)
            {
                for (int j = 0; j < r; j++)
                {
                    cout << row[i] << col[j] << " ";
                }
                cout << endl;
            }
            cout << endl;
            minorCount++;
        }
    }

    cout << endl
         << "Minor count : " << minorCount << endl;
}
```


Parallel implementation

```
long int getMinors(long int combinations[], int n, int r, int world_rank, int world_size, mat_ZZ_p mat)
{
    long int globalZeroCnt = 0;
    long int col[r], row[r];
    long int combinationCount = nCr(n, r);
    long int totalCombinations = pow(combinationCount, 2);
    long int blockSize = totalCombinations / world_size;
    long int startRow = (world_rank * blockSize) / combinationCount;
    long int startCol = (world_rank * blockSize) % combinationCount;

    if ((totalCombinations < world_size) && (world_rank < totalCombinations))
    {
        blockSize = 1;
        startRow = world_rank / combinationCount;
        startCol = world_rank % combinationCount;
    }

    if (world_rank == (world_size - 1) && (totalCombinations >= world_size))
    {
        long int excessRows = totalCombinations % world_size;
        blockSize += excessRows;
    }
}
```

```

long int count = 0;
get_kth_combination(combinations, n, r, startRow, row);

while (count < blockSize)
{
    get_kth_combination(combinations, n, r, startCol % combinationCount, col);

    globalZeroCnt = minorDeterminant(r, row, col, mat, world_rank);

    startCol++;
    if (startCol == (combinationCount) && startRow < (combinationCount - 1))
    {
        startCol = 0;
        startRow++;
        get_kth_combination(combinations, n, r, startRow, row);
    }

    count++;
}

return globalZeroCnt;
}

```

The program has been tried out on a matrix of dimension

- Dimension of the main matrix : 50×50
Dimension of the minor : 6×6
Minor count : 252514346490000
Number of processors : 20
Program execution time : More than 1 hour.
-

- Dimension of the main matrix : 50×50
Dimension of the minor : 2×2
Minor count : 1500625
Number of processors : 20
Program execution time : Around 6 minutes.

n	1x1	2x2	3x3	4x4	5x5	6x6	7x7	8x8	9x9	10x10
50	2500	1500625	384160000	53038090000	4489143937600	252514346490000	9.97689E+15	2.88239E+17	6.2772E+18	1.0552E+20

Setting up Master-Slave architecture on the cluster

- Total available servers (in working condition) : 6
- Total number of processors : 48 (8 processors available on each server)
- Set up 1 node in the cluster as the master, and other 5 as slave nodes.

Setup steps :

1. Adding a new privileged user - 'mpiuser', on each node.
2. Installing necessary packages (OpenMPI, NTL, nfs, openssh, rsync) on each node.
3. Enabling password-less ssh between each slave node and the master node.
4. Configuring 'nfs' and 'rsync' on each node to load real time changes in the shared directory on the master.

Things I have learned till date

- Double and Add algorithm along with what ECDLP means.
- Integrating NTL and OpenMPI : parallelizing minor retrieval from the matrix.
- Using various functions of OpenMPI like MPI_Barrier(), MPI_Bcast(), MPI_Finalize(), MPI_Comm_size(), MPI_Comm_rank()
- Learning basics of NTL (Number Theory Library) in C++ : getting familiar by using datatypes like ZZ_p, mat_ZZ_p, and functions like NumRows(), det(), etc.
- Basics of what Finite Fields are.
- Configuration of Master-Slave architecture in a cluster containing 6 working servers using password-less ssh, nfs, rsync.

Thank You