**Geo-Encryption Algorithm**

***Abstract* - Data Security is one of the biggest concerns of the free world. Cyber-warfare and cyber-attacks are now more dangerous for a country or an organisation than any other form of attacks. Data protection has now become important more than ever. With the recent advances in the field of supercomputers and the attacks that are being made to crack the existing algorithms like the Advanced Encryption Algorithm (AES), this paper proposes a new algorithm that will randomise the data to arbitrary points on a 2D plane and generate a random key for each point. The algorithm proposed is fast and more secure than the currently used encryption algorithms.**

***1. Introduction***

Cryptography is the study of such practices or algorithms that will allow communication between two parties in a secure manner, such that any intermediary malicious party isn’t able to trace or get their hands on the original data that was being communicated. Cryptologists design various types of algorithms keeping data confidentiality, data integrity, authentication and non-repudiation in their mind. A cryptographic system comprises of a sender who has a data (Plain text) that is encrypted on the senders’ side into an unreadable format (Cipher text) and then sent to the intended receiver who uses an algorithm or a unique key to decrypt the cipher text to the plain text and hence retrieving the original data back. Professionals have the skill in production cryptanalysis but innovation in the design of new types of cryptographic systems has been amateurs’ forte (Diffie W. et al. 1976). On that note the paper proposes a new Geo – Encryption algorithm. It is a symmetric block algorithm that can be used to encrypt data to an incomprehensible format using a key and then decrypt the data using the same key to get our original data back. We also use

2.5 Design Decisions

No weak keys as a design goal

**4. Building Blocks**

Boolean operations like AND, OR and XOR are analogous to set operations. AND mathematically represents intersection, OR represents union and XOR - the difference between the inputs. This is called the isomorphic nature of XOR which allows to toggle between the inputs.

If C = (A ⊕ B)

then, A = (B ⊕ C) = B ⊕ (A ⊕ B)

and B = (A ⊕ C) = A ⊕ (A ⊕ B)

So, we can get the value of A if we have the values of B and (A ⊕ B).

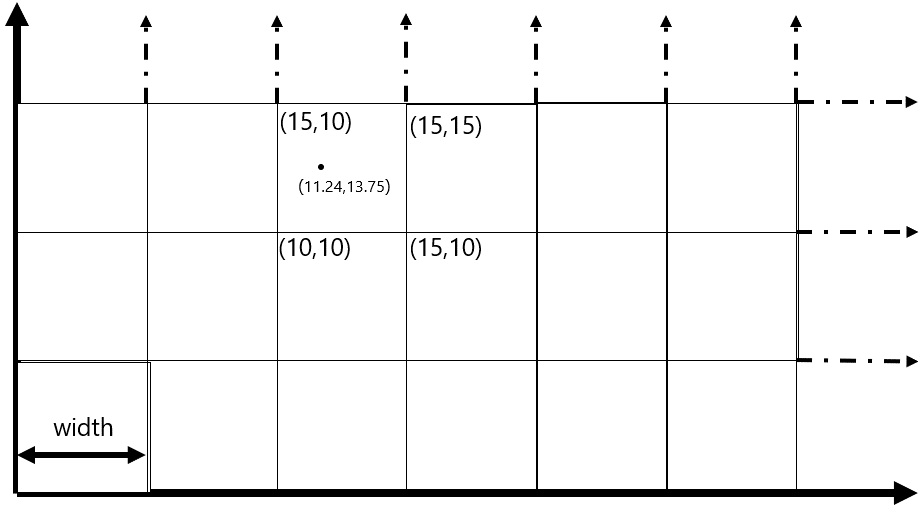


Figure 1: Cells of the Grid

**5. Description of the Algorithm**

*Configurations –*

This algorithm needs the encrypting party and the decrypting party to agree upon some configuration options, as below.

Block size (*l*): The length of the block in bits, to be treated as one entity while encrypting.

Block Value (V): Value of *l*-bit block on radix 10.

Width (w): The width of the cell.

Key (K): Random bit array of size k.

Key Length (k): The length of key in bits.

Code-Book (CB): A 2-D array index on the x-min and y-min of each cell and stores the key associated with the cell.

Block Value Mapper Bits (BVMB): It will be half the size of the key in bits.

Key Mapper Bits (KMB): This will decide the key.

*Encryption -*

The most used algorithms today such as, AES, Elliptic Curve, 3DES, etc all have a basic idea of replacing a character by another and/or use prime numbers in computation process. This makes these algorithms computationally heavy and time taking. The idea proposed here is different in these aspects, i.e. neither does it map a character to another, nor does it use prime numbers.

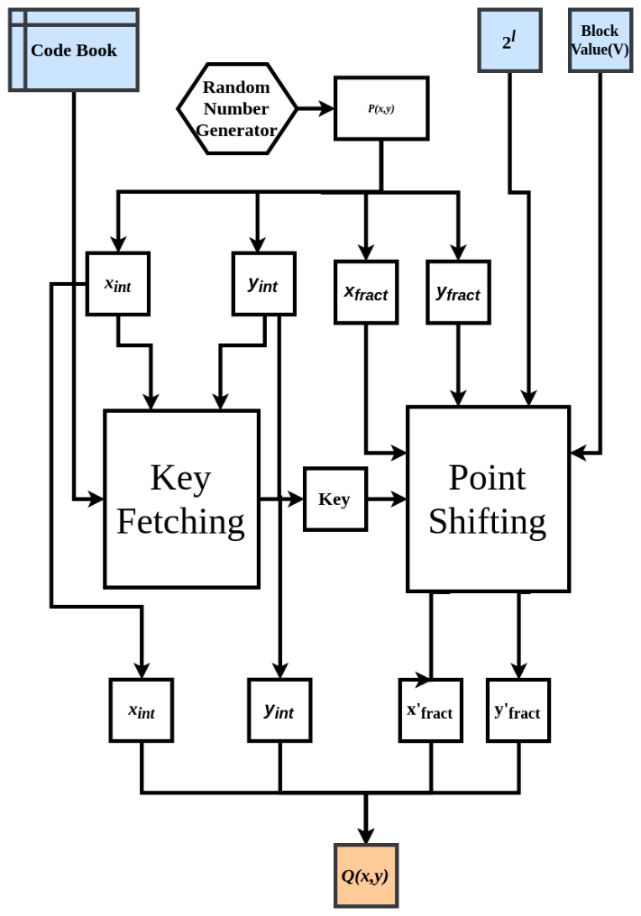


Figure 2: Encryption

If the encryption function is represented mathematically, one gets the bellow equation.

f(b): B→G

where,

B: A set of blocks of size *l.*

G: The X-Y plane with total cells *n* and width *w*.

When describing B, a set of blocks of a given block size infers that B is a set of bit arrays, each of length *l*. For example, if *l* =32, B is a set that comprises all the values from 0 to 232 - 1 in binary, formatted to 32bits with leading zeroes. G can be viewed as a plane with grid as shown in Fig 1. Each cell of this grid has associated with itself a random key. The key length (k) in bits is twice the size of block (*l*). All the points in each cell will only use the associated key of that cell while encryption and decryption.

k = 2 \* *l*

Where, k: Key length *l*: Block size

The Point (P) which would be represented using (*x, y*) would have four values. The integer part of *x* and *y*, and the fractional part of *x* and *y*. As the algorithm maps a block to a point P (*x*, *y*), the encryption process can simply be described as “To find a random point that maps to the desired block value”. To achieve this, a random point is selected and then shifted such that it represents the desired block value. As the randomly selected point decides the key value it is necessary that the point does not shift to another cell, and in-process change the key. So, while shifting it is noted that only the fractional part changes leaving the integer part intact. Due to this the fractional part is ignored while fetching the key from the Code-Book and integer part is ignored while shifting. As the key fetching and the point shifting depends upon distinct parts of *x* and *y*, point P can be represent as,

*P (xint, xfract, yint, yfract)*

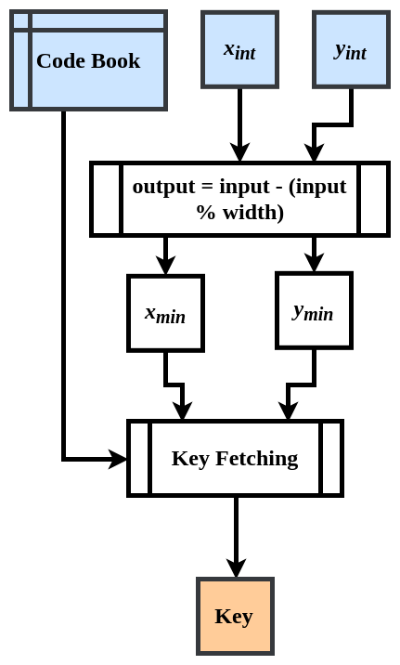
**

Figure 3: Fetching the Key

*Key Fetching –*

To fetch the key, we need to find the cell in which the randomly selected point lies. As the grid is of uniform width (w) and the Code-Book is indexed on the minima of each cell, the key can be located as,

K = CB [*xint* – (*xint* mod w)] [*yint* – (*yint* mod w)]

Now this key is used in encrypting the upcoming block of the data. The fractional part of the x and y are also random points which are concatenated together and then a XOR operation is performed on them with the key. Length of α is going to be same as that of the key.

α = (fract(x) ● fract(y))

β = α ⊕ Key

The α value needs to be adjusted such that it represents the block value to be encoded. So, on adding the offset to β and again operating a XOR operation on it with the key,

γ = β mod 2*l*

offset = γ – V

β*'* = β + offset

α*'* = β*'* ⊕ Key

it gives a value α', which is then split into two values. The first half represents *x'fract* and the second half represents *y'fract*. These bits have the encoded data in them.

*x'fract* ● *y'fract* = α*'*

The *x'fract* and *y'fract* values are now merged with the integer parts of x and y that were selected randomly in the very beginning of the algorithm, thus retaining the key location on the grid and hiding the intended data.

*P(xint,xfract,yint,yfract)****→****Q(xint,x'fract,yint,y'fract)*

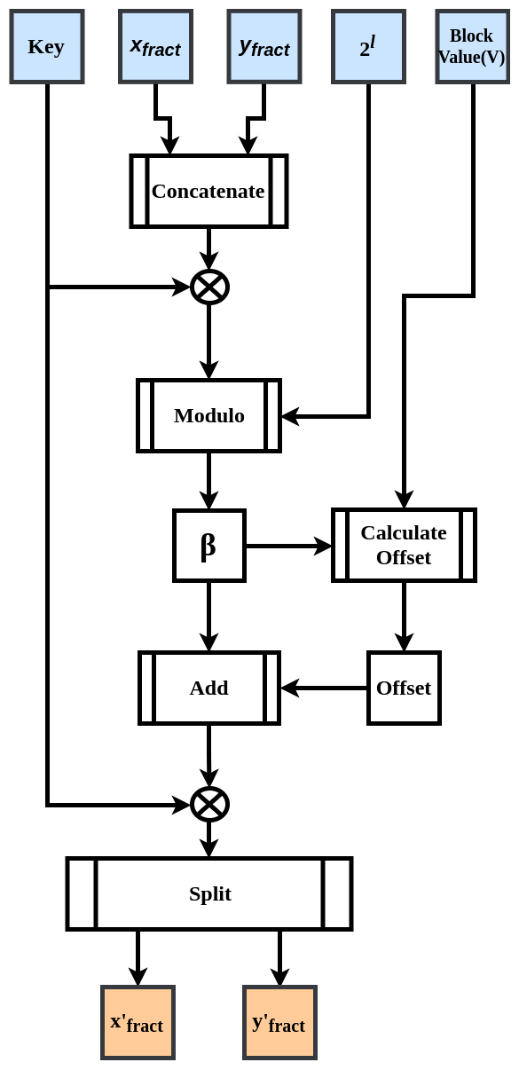


Figure 4: Shifting the Point P to Point Q

Each block is written serially in its encrypted format in a binary file which can then be shared over the network to the intended receiver (decrypting party).

*Decryption –*

The decryption process starts with the encrypted point (Q) as input. As the integer portion of the Point P does not change while encryption, the key-fetching for decryption can be done in the same way as in the encryption process. Once the key is fetched, the block value can be retrieved from the fractional portions of Point *Q(xint,x'fract,yint,y'fract)* by backtracking.

α = (*x'fract* ● *y'fract*)

β = α ⊕ Key

V = β mod 2*l*

V here is the block value that the sender intended to send to the receiver. The block bits can be obtained by formatting V to *l* bits size with leading zeroes.

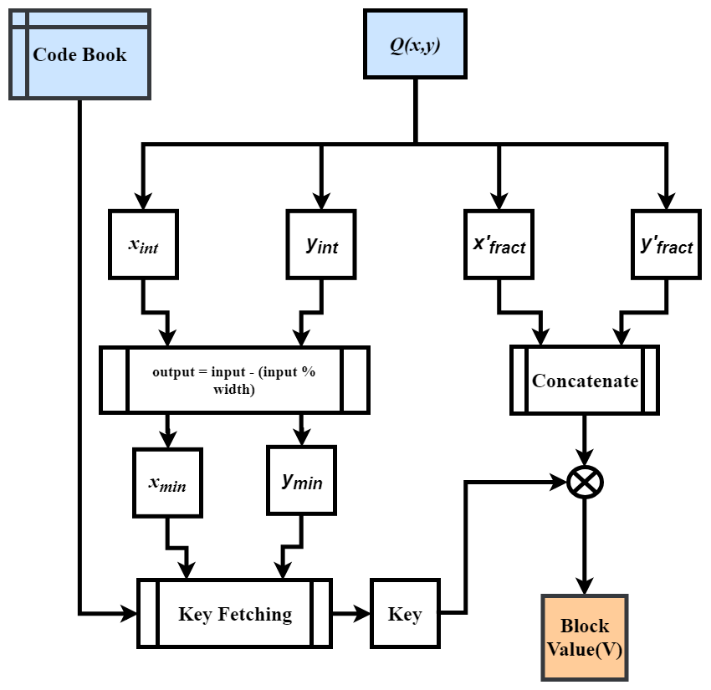


Figure 5: Decryption Process

**6. Discussion**

*6.1 Padding*

In an ideal case, each file will have file size divisible by block of size; but in real time scenario, this is rarely the case. For the algorithm to work on any file size, a padding needs to be appended in the encrypted file. In this case the padding is the number of bytes in the last block.

For example, lets consider a file size of 1029 bytes to be encrypted with block size of 8 bytes (*l =* 64 bits), then the last block will have 5 bytes of data. This makes the padding value to be 5. This value is appended in the beginning of the encrypted file. So, during decryption the encrypted file is read, and this value is used to cut short the last block of the decrypted data appropriately.

6.2 *Random Number Generator*

In any encryption algorithm, randomness/entropy is of prime importance. Many implementations of Random number generators are not purely random and so are prone to attacks. While key and point generation, it is necessary that proper randomness is ensured. During testing of the proposed algorithm, Pythons Secret library is used for the key generation and the boost library of C++11 is used get random Points.

6.3 *Parallelization*

As each block uses a distinct key, there is no need for chaining as is in AES and other algorithms. The encryption process of each block is isolated from other blocks. This makes the algorithm highly parallelizable. While testing, OpenMP of C++ was used for parallelising the algorithm which increased the performance two times on a dual core CPU.

6.4 *Attacks*

6.4.1 *Brute Force Attack*

For a 64 bit block size the key size is 128 bits. This makes the total number of possible keys to be 2128 which is around 3.4 x 1038. Furthermore, each cell has its own 128 bit distinct key assigned to it, which makes the number of possible combinations to be,

Where n is the number of total cells in the grid G.  
Hence making the computational complexity of breaking the algorithm very high.

6.4.2 *Known Plain Text Attack*

As each block is mapped on different points of different cells which have different keys associated with them, known plain-text attacks should also not be able to predict the key. The algorithm will be only partially breached even if many keys are predicted somehow. As all the blocks can be mapped to any of the cells therefore, they cannot be associated to any specific defined region of the Grid.

6.5 *Performance Measure*

As discussed above, the algorithm is parallelisable and does not use prime factorization, this makes the algorithm computationally fast. When tested on a machine with 64-bit, i7 processor with 4 logical processors, 6500U CPU @ 2.5GHz, 8GB RAM, the results obtained were as shown in Table 1 below.

Table 1: Performance Measure

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| File Size (KB) | Max no. of keys used per iterations | No. of iterations | Total Time Taken | Performance (KB/s) |
| 1 | 128 | 100000 | 19.727 | 5069.19 |
| 10 | 1280 | 10000 | 16.7011 | 5987.62 |
| 100 | 12800 | 1000 | 15.9831 | 6256.60 |
| 1000 | 128000 | 100 | 16.1519 | 6191.22 |
| 10000 | 1280000 | 10 | 16.3845 | 6103.32 |
| 100000 | 12800000 | 1 | 16.9779 | 5890.00 |

While testing file sizes of 1KB to 106KB (~100MB) were encrypted several times to calculate an average performance. The second column in Table 1 describes the maximum number of keys that the algorithm will take for encrypting the file once. Thus, for a 1KB file, around 128 keys each of size 128-bits will be used.

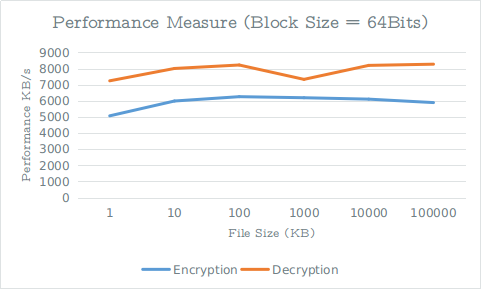


Figure 6: Performance Measure where l = 64

As Figure 6 and 7 depict, the decryption performance is better than encryption performance. Though there is not a major difference but as file size increases the difference becomes observable.

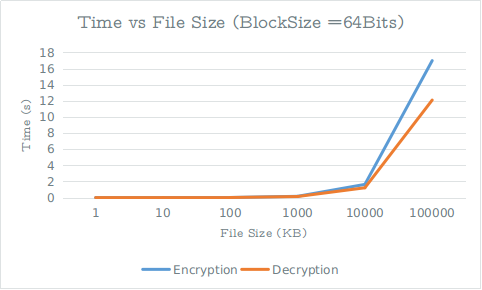


Figure 7: Time taken vs file size

The performance with 32-bit and 64-bit block size on a 64-bit, i5 processor with 4 logical processors, 5200U CPU @ 2.2GHz, 8GB RAM was as shown in Table 2 below. As we can see that increasing the block size does not affect much on the performance but increase the security two times.

Table 2: Performance(KB/s) when l is 32 and 64

|  |  |  |
| --- | --- | --- |
| Block Size | 32-bit | 64-bit |
| Encryption | 4393.69 | 4589.32 |
| Decryption | 6228.82 | 6064.17 |

**7. References**

https://www.eetimes.com/document.asp?doc\_id=1279619#