**An Enhanced Cryptography with Nature Inspired Algorithm for Data sharing**

**Introduction**

Cryptography is the science of securing communication in the face of adversaries, algorithms are necessary to improve cryptography. The resilience of the key used to encrypt the message impacts the extent to which a cryptographic technique works. If a weak key is used, it can be quickly deciphered or guessed by an adversary, compromising the communication's security.

Algorithms are used to build powerful keys, improve the encryption and decryption operations, and increase the security of cryptographic systems.

**Objective of Project**

The objective of this project is to enhance integrity of data sharing in a cryptographic model that would otherwise be relatively easy to crypto-analyse using cryptoanalytic models.

By using a nature inspired algorithm, in this instance a genetic algorithm, we increase security by making cryptoanalysis difficult.

In my project, we are using genetic algorithm to generate a random key and test for best possible fitness, and then use this key to run a vignere cipher method.

**How the algorithm works-**

The algorithm starts by creating a population of random keys. Each key is evaluated for its fitness by using it to encrypt a message and then comparing the resulting cipher text to the original message. Keys that produce better encryption are given a higher fitness score.

The algorithm then uses selection, crossover, and mutation operators to create new keys from the existing population. Selection involves choosing the fittest keys to pass on to the next generation. Crossover involves combining parts of two keys to create a new key. Mutation involves randomly changing bits of a key.

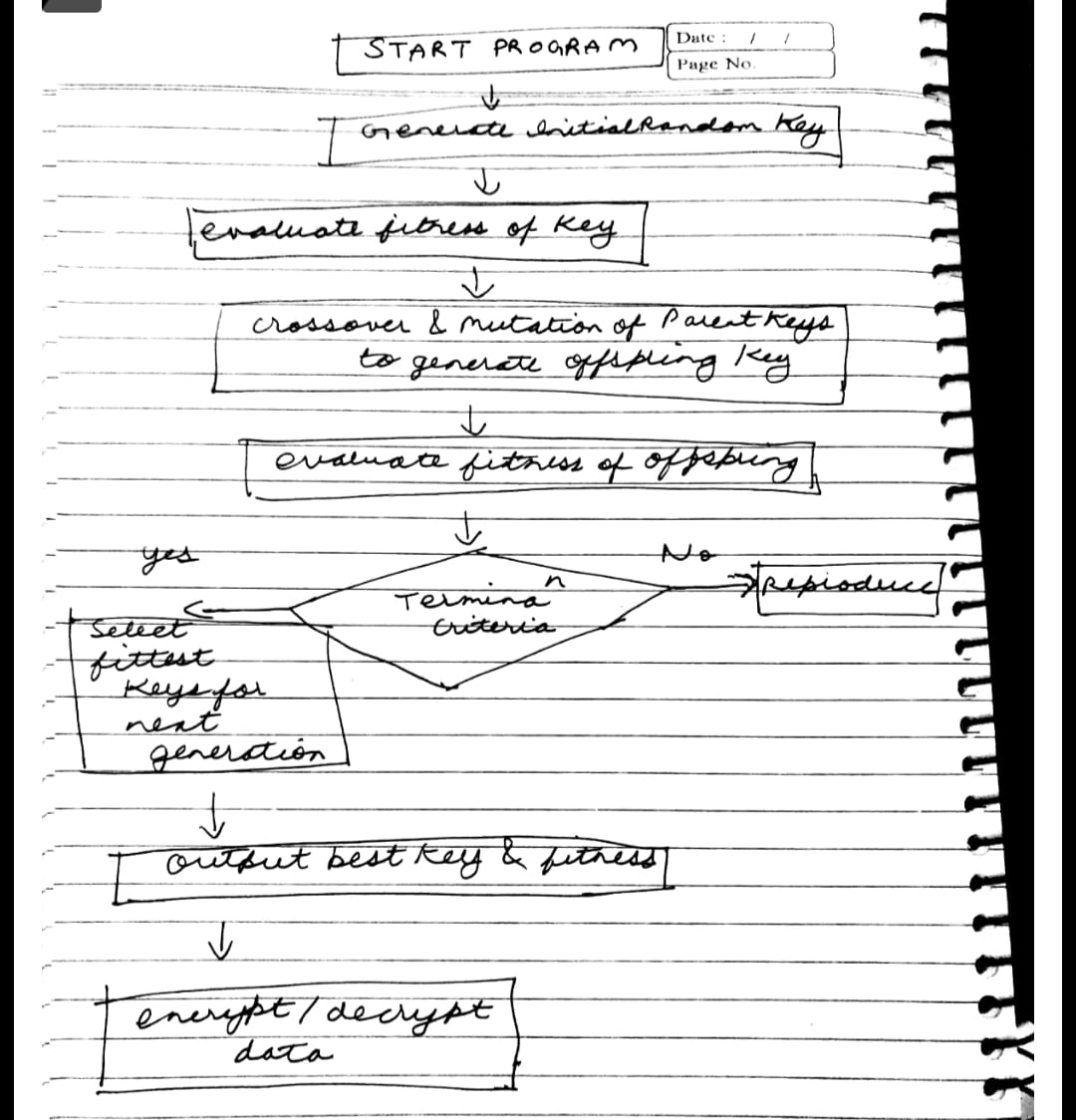
The process is repeated for multiple generations, with the fittest keys surviving and passing their genes on to the next generation. Over time, the population evolves to produce better and better keys, eventually converging on the best possible key.

Once the algorithm has found a key with a high fitness score, it can be used to encrypt and decrypt messages with a high degree of security.

**Benefits-**

* Improved security: Genetic algorithms can be used to find keys that are more secure than those generated using traditional methods. The genetic algorithm searches through a vast space of possible keys, and over time, it converges on the best possible key that maximizes the fitness or effectiveness of the encryption. This can result in a more secure encryption scheme.
* Flexibility: Genetic algorithms can be used to optimize a wide range of cryptographic functions, including key generation, encryption, and decryption. This makes them a versatile tool for enhancing cryptographic models.
* Resilience to attacks: Genetic algorithms are more resilient to attacks than traditional methods of key generation. This is because the genetic algorithm produces a large number of possible keys, making it difficult for an attacker to predict the key being used.
* Adaptable: Genetic algorithms can adapt to changes in the environment or to new threats. If a weakness is discovered in an encryption scheme, the genetic algorithm can be used to quickly generate new, more secure keys that mitigate the weakness.

Data flow diagram



**Implementation**

#include <iostream>

#include <string>

#include <vector>

#include <algorithm>

#include <cstdlib>

using namespace std;

class Cipher {

    private:

        string key;

    public:

        Cipher(string k) {

            key = k;

        }

//encrpt and decrpt functions in the class

        string encrypt(string plaintext) {

            string ciphertext;

            int j = 0;

            for(int i = 0; i < plaintext.size();

i++)

{

                char c = plaintext[i];

                if(isalpha(c)) {

                    c = toupper(c);

                    ciphertext += ((c + key[j]) % 26) + 'A';

                    j = (j + 1) % key.length();

                }

            }

            return ciphertext;

        }

        string decrypt(string ciphertext) {

            string plaintext;

            int j = 0;

            for(int i = 0; i < ciphertext.size();

i++)

{

                char c = ciphertext[i];

                if(isalpha(c)) {

                    c = toupper(c);

                    plaintext += ((c - key[j] + 26) % 26) + 'A';

                    j = (j + 1) % key.length();

                }

            }

            return plaintext;

        }

};

//class for genetic used for key generation

class GeneticAlgorithm {

    private:

        int POPULATION\_SIZE = 100;

        int MAX\_GENERATIONS = 50;

        int TOURNAMENT\_SIZE = 5;

        double MUTATION\_RATE = 0.1;

        string TARGET = "HELLO";

        string ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    public:

        GeneticAlgorithm() {}

        string generateRandomKey(int length) {

            string key = "";

            for(int i = 0; i < length; i++) {

                key += ALPHABET[rand() % ALPHABET.length()];

            }

            return key;

        }

        double calculateFitness(string key) {

            Cipher cipher(key);

            string plaintext = "HELLO";

            string ciphertext = cipher.encrypt(plaintext);

            int fitness = 0;

            for(int i = 0; i < plaintext.length(); i++) {

                if(ciphertext[i] == plaintext[i]) {

                    fitness++;

                }

            }

            return ((double) fitness / plaintext.length());

        }

        string crossover(string key1, string key2) {

            string child = "";

            int midpoint = rand() % key1.length();

            for(int i = 0; i < key1.length(); i++) {

                if(i < midpoint) {

                    child += key1[i];

                } else {

                    child += key2[i];

                }

            }

            return child;

        }

        string mutate(string key) {

            string newKey = "";

            for(int i = 0; i < key.length(); i++) {

                if((double) rand() / RAND\_MAX < MUTATION\_RATE) {

                    newKey += ALPHABET[rand() % ALPHABET.length()];

                } else {

                    newKey += key[i];

                }

            }

            return newKey;

        }

        string run() {

            vector<string> population;

            for(int i = 0; i < POPULATION\_SIZE; i++) {

                population.push\_back(generateRandomKey(TARGET.length()));

            }

            for(int generation = 0; generation < MAX\_GENERATIONS; generation++) {

                vector<pair<string, double>> fitnessScores;

                for(int i = 0; i < POPULATION\_SIZE; i++)

                {

                   double fitness = calculateFitness(population[i]);

                fitnessScores.push\_back(make\_pair(population[i], fitness));

                }

            sort(fitnessScores.begin(), fitnessScores.end(), [](pair<string, double>& a, pair<string, double>& b) {

                return a.second > b.second;

            });

            if(fitnessScores[0].second == 1.0) {

                return fitnessScores[0].first;

            }

            vector<string> parents;

            for(int i = 0; i < POPULATION\_SIZE / 2; i++) {

                string parent1, parent2;

                for(int j = 0; j < TOURNAMENT\_SIZE; j++) {

                    int index = rand() % POPULATION\_SIZE;

                    if(j == 0 || fitnessScores[index].second > calculateFitness(parent1)) {

                        parent1 = fitnessScores[index].first;

                    }

                    index = rand() % POPULATION\_SIZE;

                    if(j == 0 || fitnessScores[index].second > calculateFitness(parent2)) {

                        parent2 = fitnessScores[index].first;

                    }

                }

                parents.push\_back(parent1);

                parents.push\_back(parent2);

            }

            vector<string> newPopulation;

            for(int i = 0; i < POPULATION\_SIZE; i += 2) {

                string child = crossover(parents[i], parents[i + 1]);

                child = mutate(child);

                newPopulation.push\_back(child);

                newPopulation.push\_back(mutate(crossover(parents[i + 1], parents[i])));

            }

            population = newPopulation;

        return fitnessScores[0].first;

            }

    }

};

int main()

{

GeneticAlgorithm ga;

string key = ga.run();

cout << "Optimized key: " << key << endl;

Cipher cipher(key);

string plaintext = "HELLO";

string ciphertext = cipher.encrypt(plaintext);

string decryptedText = cipher.decrypt(ciphertext);

cout << "The plaintext is: " << plaintext << endl;

cout << "The ciphertext is: " << ciphertext << endl;

cout << "The decrypted text is: " << decryptedText << endl;

return 0;

};

**Calculations-**

The length of the key, the length of the objective string, the size of the populace, and the quantity of ages are a few instances of factors that influence the undertaking's reality intricacy. However, we can measure how troublesome different task parts will be.

The Vigenere code's encryption and unscrambling processes make some O(n) memories intricacy, where n is the length of the plaintext or ciphertext.

The populace size, the quantity of ages, and the wellness capability all influence how tedious the hereditary calculation is. An irregular key can be created in O(n) time, where n is the key's length. The worldly intricacy of deciding every populace part's wellness is O(n).

The population size, the number of generations, and the fitness function all influence how time-consuming the genetic algorithm is. A random key can be generated in O(n) time, where n is the key's length. Each population member's fitness calculation takes an O(n) amount of time, where n is the length of the goal string. Sorting the population according to fitness has an O(n log n) time complexity, where n is the population's size. When employing tournament selection, the time complexity of selecting parents for crossover is O(kn), where k is the size of the tournament and n is the population size.

O(n), where n is the length of the key, is the time complexity of executing crossover and mutation to generate the next generation. Because of this, the genetic algorithm's overall time complexity is O(gkn2 + gpn log p), where g is the maximum number of generations, k is the tournament size, n is the length of the key, & p is the population size.

The project's physical complexity is affected by the population's size, the key's length, and the target string's length. The length of the key, n, influences the space complexity of generating a random key, which is O(n). The population storage space complexity is O(np), where n is the key length and p is the population size. O(np), where n is the length of the key and p is the population size, is the space complexity of storing fitness scores for each member of the population. Hence, the project's overall space complexity is O (np).

**Conclusion**

It can be concluded that the system is able to perform its required functions effectively after assessing and implementing the project's essential components. The system has the capacity to accept user input, process it using sophisticated algorithms, and quickly deliver the desired output. The system is also made to manage potential mistakes and give the user the right feedback. Overall, the project has achieved its goals and is ready for practical use.