Description

BACKGROUND OF THE DISCLOSURE

This invention is related to computer science. More specyfic it is related to crypthographic communications.

Information in many modern electronic systems required to be encrypted. This is used to ensure only parties predetermined, will know message. In case of symetric cyphers both sender and recipent of message must know some secret information called key. This key can be used to read message previously encrypted with same key. This is needed as some third party may access message. To prevent that party to read message, it need to be in form that cannot be easly understood. This is purpose of encryption.

Encryption may ensure privacy as well as security. It requires that both parties: sender and reciver first share some secret called key and written as number. This secret key is first used to encrypt and later to decrypt message. Hovewer secure transmission of keys is seperate problem with already many proposed solutions.

R. Rivest, A. Stein and L. Adleman proposed RSA communication method and system that is often used to share key between two parties. Another common solution is to provide key on other secure chanel of communication like for example in person.

SUMMARY OF THE INVENTION

Given is message consisting of n numbers M_1, M_2, … M_n. There is also key consisting of o numbers K_1 K_2 … K_o. System use random number generator. Number random generator is such method that allows to generate random that cannot be easly assumed beforehand. Simple exampel is rolling dice – we never can tell easly what number we will get. Modern random number generators are easly and widely available for programmers. Computers with intel processors provide random_device method in C++ programming language that generates random numbers. System also use pseudo random generator. Pseudorandom generator generates numbers that are hard to be predicted not knowing initial state of generator. Hovewer there can be determined with 100% certanity if necesery information about initial state is given. One of popular software method for pseudo random number generator is Mersenne Twister proposed in 1997 by Makoto Matsumoto and Takuji Nashimura.

Key in system is used to setup initial state of pseudorandom generator. Later generator is used to generate m numbers. Each generated number must be in range 0 to total number of already generated numbers plus n. Thus first number must be in range 0 and n. Second generated number must be in range 0 to n+1. Third generated number must be in range 0 to n+2 etc.
Numbers generated by pseudorandom generator will be called indexes. Later system use random number generator to generate n random numbers. Those numbers will be called *hidden*.
First number M_1 is xored with first hidden number. Second number M_2 is xored with second hidden number and so on for all numbers M. Then additional (m-n) random numbers are generated called *noise*. First *noise* number is inserted in position determined by first index number. Second *noise* number is inserted in position determined by second index number. This is done for all *noise* numbers. Once all *noise* numbers are inserted into message, all *hidden* numbers are inserted into message. Number m need to be much larger than number n to ensure encrypted message wont introduce any significant pattern into series of mostly random numbers. Such prepared message is encrypted ciphertext. It can be send in that form to recipent by sender.

Once recipent gets message he or she must also use key to initialise state of pseudorandom generator. It must be same pseudorandom generator sender used to ensure it will generate same numbers if initialised into same state. Later generator is used to generate m numbers.Numbers generated by pseudorandom generator will be called indexes. Number on position estimated by last index is removed. Later number on position estimated by index one before last is removed.
This way all numbers inserted previously as *noise* are being removed. Last n removed noise numbers need to be stored. Once they are stored they are used to xor with remaning message.

IMPLEMENTATION

Below is implementation of encryption method usde to encrypt message and decryption method used to read encrypted message.Implementation have been writen in Visual Studio 2017 in C++ programming language. Implementation use come arbitrary chosen maximum_text_length.
It uses 1000 times more noise symbols to make message around 0,1% of all symbols in cipher and thus render its effect as insignificant on any statistical properties that could help to break code.
Implementation use pseurodarandom number generator called mersene twister. Hovewer instead of decribing state as initial state after creation of mersene tiwster object with some part of key it use it state after inserting given part number and later set it state to net part of key.

```
#ifndef WATERDOWN_H
#define WATERDOWN_H

#include <iostream>
#include <vector>
#include <random>
#include <time.h>

using namespace std;

namespace waterdown {
	//Encryption is process of converting data to unrecognizable form
	typedef unsigned int symbol;
	const size_t maximum_text_length = 90*1024;
	const size_t password_length = 64; //this is in quadruplets of chars that is 64 for 
example means 4*64 = 256 char long password
	const size_t extended_size = 90* 1024*1024;
	symbol random_char() {
		std::random_device rd;
		(symbol)return rd();
	}

	bool encrypt(vector<symbol>& plain_text, vector<symbol>& encrypted_text, int 
key[password_length + 1]) {
		if (plain_text.size() > maximum_text_length) {
			return false;
		}

		const size_t part_size = extended_size / password_length;
		const size_t text_length = plain_text.size();

		vector<symbol> mask(text_length);
		for (int i = 0;i < text_length;++i) {
			mask[i] = random_char();
		}

		try {
			encrypted_text.resize(extended_size);
		}
		catch (std::bad_alloc& wrong_alloc) {
			std::cerr << "bad_alloc caught: " << wrong_alloc.what() << '\n';
			return false;
		}
```

```
        if (encrypted_text.size() != extended_size) {
                return false;
        }

        encrypted_text.resize(0);
        encrypted_text.reserve(extended_size);
        encrypted_text = plain_text;
        for (int i = 0;i < encrypted_text.size();++i) {
                encrypted_text[i] ^= mask[i];
        }
        int up = 0;

        for (int key_part_index = 0;key_part_index < password_length;++key_part_index)
        {
                mt19937 mt_rand(key[key_part_index]);
                for (int i = 0;i < part_size; ++i) {
                        std::uniform_int_distribution<int> dis(0, up + text_length);
                        int insert_index = dis(mt_rand);

                        symbol rchar = random_char();
                        try {
                                encrypted_text.insert(
                                encrypted_text.begin() + insert_index, rchar);
                        }
                        catch (std::bad_alloc& wrong_alloc) {
                                std::cerr << "bad_alloc caught: " <<
                                wrong_alloc.what() << '\n';
                                return false;
                        }
                        ++up;
                }
        }

        mt19937 mt_rand(key[password_length]);
        for (int ins = 0;ins < text_length;++ins) {
                std::uniform_int_distribution<int> dis(0, up + text_length);
                int insert_index = dis(mt_rand);
                encrypted_text.insert(
                        encrypted_text.begin() + insert_index, mask[ins]);
                ++up;
        }

        return true;
}

bool decrypt(vector<symbol>& encrypted_text, int key[password_length + 1]) {
        const size_t part_size = extended_size / password_length;
        vector<int> index;
        size_t text_length = encrypted_text.size();
        text_length -= extended_size;
        text_length /= 2;
        try {
                index.resize(part_size);
        }
        catch (std::bad_alloc& wrong_alloc) {
                std::cerr << "bad_alloc caught: " << wrong_alloc.what() << '\n';
                return false;
        }
        int up = 0;
        vector<int> indexes_elements_to_be_removed;
        vector<symbol> mask(text_length);
        for (int key_part_index = 0; key_part_index < password_length; +
                                                +key_part_index) {
                mt19937 mt_rand(key[key_part_index]);
```

```
                for (int i = 0;i < part_size; ++i) {
                        std::uniform_int_distribution<int> dis(0, up + text_length);
                        int insert_index = dis(mt_rand);
                        indexes_elements_to_be_removed.push_back(insert_index);
                        ++up;
                }
        }

        mt19937 mt_rand(key[password_length]);

        vector<int>mask_location(text_length);
        for (int ins = 0;ins < text_length;++ins) {
                std::uniform_int_distribution<int> dis(0, up + text_length);
                int insert_index = dis(mt_rand);
                mask_location[ins] = insert_index;
                ++up;
        }
        for (int rem = (text_length - 1);rem >= 0;--rem) {
                mask[rem] = encrypted_text[mask_location[rem]];
                encrypted_text.erase(encrypted_text.begin() + mask_location[rem]);
        }
        for (int i = indexes_elements_to_be_removed.size() - 1;i >= 0;--i) {
                if (indexes_elements_to_be_removed[i] > encrypted_text.size()) {
                        cout << "Error : " << encrypted_text.size() << " " <<
                        indexes_elements_to_be_removed[i] << " " << i;
                        return false;
                }
                encrypted_text.erase(encrypted_text.begin() +
                indexes_elements_to_be_removed[i]);
        }

        for (int k = 0;k < text_length;++k) {
                encrypted_text[k] ^= mask[k];
        }
        return true;
    }

}

#endif
```

DRAWINGS

Fig. 1. Presents text „ABC" before encryption.

Fig. 2. Presents text after inserting random number on position with index 1. Random number is denoted as blank field as it can be any number.

Fig. 3 Presents text after inserting random number on position with index 3 after previously inserting random number on position 1. This is to show that the more random numbers will be inserted the more difficult it will be to find text before encryption.

Fig 4. Presents general overwier. Random number generator is used to generate random numbers. Pseudorandom number generator is to establish order in which those random numbers will be inserted into text that need to be encrypted. After inserting random numbers into text it will lose its orginal format and structure. Ilustration further highlights that if only few random number would be inserted orginal structure of message could still be deduced. This is ilustrated by using similiar cloud graphics for both text and encrypted cipher.