UNIVERSITÉ DU QUÉBEC À MONTRÉAL

CRYPTOGRAPHIE N-DIMENSIONS AUTO-CORRIGÉE

TRAVAIL

PRÉSENTÉ
COMME EXIGENCE PARTIELLE

INFORMATIQUE ET GÉNIE LOGICIEL

PAR

MAXIM THIBODEAU

NOVEMBRE 2024

AVANT-PROPOS

Dans le cadre du programme d'informatique et génie logiciel, il nous est donné l’opportunité de nous pencher sur des problématiques aillant plus à trait à la sphère de la technique. Cet article se veut révolutionnaire, car je n'ai pas eu l'opportunité de voir cette combinaison d'idées nul part.

TABLE DES MATIÈRES

Table des matières

AVANT-PROPOS....2

TABLE DES MATIÈRES....3

LISTE DES ABRÉVIATIONS, DES SIGLES ET DES ACRONYMES....5

LISTE DES SYMBOLES ET DES UNITÉS....5

ABSTRACT....6

INTRODUCTION....7

DÉVELOPPEMENT....8

CONCLUSION....9

ANNEXE....10

BIBLIOGRAPHIE....12

Liste des figures

Liste des tableaux

LISTE DES ABRÉVIATIONS, DES SIGLES ET DES ACRONYMES

LISTE DES SYMBOLES ET DES UNITÉS

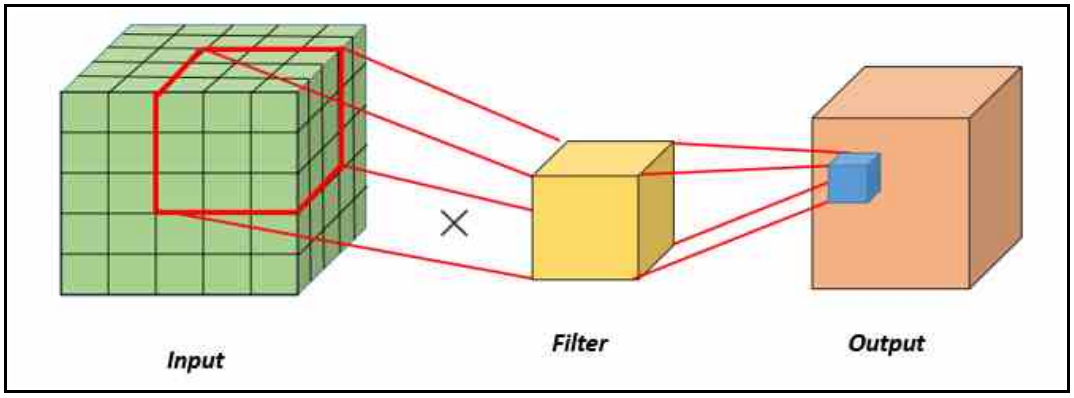
ABSTRACT

Abstract :

Les données et la sécurité de celles-ci sont d'une extrême importance, à l'heure actuelle. Cette méthode que je vais vous présenté est basé sur le repliement dans n-dimensions d'une chaîne de caractères linéaire. À la suite de cette opération, une convolution n-D est appliqué pour obtenir un référentiel linéairement dépendant à la somme des facteurs, prédétermines dans une clef n-D, et de leurs composantes associés dans la chaîne repliée. LE TOUT AUTO CORRIGÉ.

Keywords : cryptographie, dimensions, données, sécurité, correction automatique

INTRODUCTION



DÉVELOPPEMENT

- Presque inviolable : une clef de, par example 4 X 4 X 4 X 4, donne 256 cases, qui engendre avec un type « unsigned char » ==> 256^{256} possibilités == 3.2317E616
- Encryption pouvant être hautement multithreads
- Temps : plusieurs centaine de fois plus lent qu'une encryptions standard 256b, une image 430 kb est de 11 secondes.
- Décryption sans clef simple thread (impossible de multithreader), 11 secondes X suppériorité du CPU X 256^{256} X facteur chance.
- Autocorrection jusqu'à environ 20% de bits défectueux (possible > 50%)

CONCLUSION

ANNEXE

```
#include<string>
#include<math.h>
#include<random>
#include<ctime>
#include<iostream>

class correcteur {

public:

    static uint64_t* encrypting(unsigned char* inputData, unsigned char* key, int dataLength) {

        bool test = true;

        int dim = key[0];

        int convolu = key[1];

        int keyLength = pow(convolu, dim);

        uint64_t* data = new uint64_t[dataLength];

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

            uint64_t a = convolution(dim, inputData, key, i, dataLength, keyLength);

            data[i] = (inputData[i] << 56) >> 56;

            bool echec = false;

            for (int j = 0; j < 8; j++) {

                if (rand() % 10 == 0) echec = true;
            }

            if(echec) data[i] = data[i]+1;

            data[i] = data[i] | (a << 8);
        }

        return data;
    }

    static unsigned char* decrypting(uint64_t* inputData, int length, unsigned char* key) {

        int dim = key[0];

        int convolu = key[1];

        unsigned char* data = new unsigned char[length];

        int keyLength = pow(convolu, dim);

        std::vector<int> erreurs;

        for (int i = length - 1; i >= 0; i--) {
```

```
            //std::cout << ((inputData[i]<<56)>>56) << "-" << std::flush;

            data[i] = deconvolution(dim, inputData, key, i, length, keyLength, erreurs);
        }

        bool possible = true;

        while (erreurs.size() > 0 && possible) {

            bool testPremier = true;

            while (testPremier) {

                testPremier = true;

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

                    if (erreurs.size() > 0) {

                        std::vector<int> tmp;

                        deconvolution(dim, inputData, key, i, length, keyLength, tmp);

                        if (tmp.size() == 0) {

                            deleteErreurs(dim, inputData, key, i, length, keyLength, erreurs);
                            testPremier = false;
                        }
                    }
                }
            }

            std::vector<int> erreursHash;

            bool testHash = true;

            int totalSize = erreurs.size();

            while (testHash) {

                int nbrHash = erreursHash.size();

                for (int i = 0; i < erreurs.size(); i++) {

                    separer(dim, inputData, key, i, length, keyLength, erreurs, erreursHash);
                }

                if (nbrHash == erreursHash.size()) testHash = false;
            }

            bool testErreurs = true;

            while (testErreurs) {

                int size = erreurs.size();
                std::vector<int> tmpE = erreurs;

                for (int i = 0; i < tmpE.size(); i++) {
```

```
                    corriger(dim, inputData, key, i, length, keyLength, tmpE, data);
                    i -= size - tmpE.size();
                    size = tmpE.size();
                }

                if (erreurs.size() == size) testErreurs = false;

                erreurs = tmpE;
            }

            if (totalSize == erreurs.size()) possible = false;
        }

        if (erreurs.size() > 0) std::cout << "erreur(s) !!!" << std::endl;

        return data;
    }

    static unsigned char* getKey(int dim, int width) {

        if (width > 1 && width <= 10 && dim > 1 && pow(width, dim) <= 1000000) {

            uint64_t count = UINT64_MAX;
            unsigned char* key = new unsigned char[(int)pow(width, dim) + 2];

            key[0] = dim;
            key[1] = width;

            for (int i = 1; i < pow(width, dim)+1; i++) {

                key[2 + i-1] = i;
            }

            // permutter();

            return key;
        }

        std::cout << "Clef impossible !!!" << std::endl;

        return nullptr;
    }

private:

    static uint64_t convolution(int dim, unsigned char* inputData, unsigned char* key, int cntr, int
dataLength, int keyLength) {

        int width = round(pow(exp(1.0), (std::log(keyLength) / dim)));
        int widthD = round(pow(exp(1.0), std::log(dataLength) / dim));

        uint64_t somme = 0;

        for (int j = pow(width, dim) - 1; j >= 0; j--) {

            int indice = cntr;
            int i = j;

            for (int k = dim; k >= 0; k--) {
```

```
                if (pow(width, k) <= i) {

                    indice += (int)(i / pow(width, k)) * pow(widthD, k);
                    i -= pow(width, k);
                }
            }

            if (indice < dataLength) {

                somme += inputData[indice] * key[2 + j];
                //std::cout << somme << "#";
            }
        }

        std::cout << somme << "/";

        return somme;
    }

    static uint64_t deconvolution(int dim, uint64_t* inputData, unsigned char* key, int cntr, int
dataLength, int keyLength, std::vector<int>& erreurs) {

        int width = round(pow(exp(1.0), (std::log(keyLength) / dim)));
        int widthD = round(pow(exp(1.0), std::log(dataLength) / dim));

        uint64_t tmp = inputData[cntr];

        uint64_t somme = (uint64_t)(tmp >> 8);
        uint8_t hash = (uint8_t)((inputData[cntr] << 56) >> 56);

        if (hash != 100) {

            int y = 0;
        }

        for (int j = pow(width, dim) - 1; j >= 0; j--) {

            int indice = cntr;
            int i = j;

            for (int k = dim; k >= 0; k--) {

                if (pow(width, k) <= i) {

                    indice += (int)(i / pow(width, k)) * pow(widthD, k);
                    i -= pow(width, k);
                }
            }

            if (indice < dataLength) {

                uint8_t valeur = (uint8_t)((inputData[indice] << 56) >> 56);

                if (j == 0) somme /= key[2 + j];
                else somme -= valeur * key[2 + j];
            }

        }

        if (hash != somme) {
```

```
            erreurs.push_back(cntr);
        }

        return hash;
    }

    static void corriger(int dim, uint64_t* inputData, unsigned char* key, int cntr, int dataLength, int
keyLength, std::vector<int>& erreurs, unsigned char* data) {

        int width = round(pow(exp(1.0), (std::log(keyLength) / dim)));
        int widthD = round(pow(exp(1.0), std::log(dataLength) / dim));

        // trouver l'opposé géométrique:

        // effectuer les translation pour sortir l'erreur

        for (int w = 1; w <= dim; w++) {

            for (int y = -1; y <= 1; y++) {

                for (int u = 1; u < width; u++) {

                    if (y != 0) {

                        int indiceOppose = erreurs[cntr];

                        int indice = 0;
                        int i = y * u * pow(width, w) - 1;

                        for (int k = dim; k >= 0; k--) {

                            if (pow(width, k) <= i) {

                                indice -= (int)(i / pow(width, k)) * pow(widthD, k);
                                i -= pow(width, k);
                            }
                        }

                        // effectuer la convolution:

                        if (indiceOppose + indice < 0) indiceOppose = 0;
                        else indiceOppose += indice;

                        uint64_t somme = inputData[indiceOppose] >> 8;
                        uint8_t hash = (uint8_t)((inputData[indiceOppose] << 56) >> 56);

                        int posPossibleErreur = -1;
                        int ii = -1;

                        for (int j = pow(width, dim) - 1; j >= 0; j--) {

                            int indice = indiceOppose;
                            int i = j;

                            for (int k = dim; k >= 0; k--) {

                                if (pow(width, k) <= i) {

                                    indice += (int)(i / pow(width, k)) * pow(widthD, k);
```

```
                                i -= pow(width, k);
                            }
                        }

                        if (indice < dataLength) {

                            if (indice != erreurs[cntr]) {

                                uint8_t valeur = (uint8_t)((inputData[indice] << 56) >> 56);

                                somme -= valeur * key[2 + j];

                                bool testR = true;

                                for (int f = 0; f < erreurs.size(); f++) {

                                    if (indice == erreurs[f]) testR = false;
                                }

                                if (!testR) posPossibleErreur = j;
                            }
                            else ii = j;
                        }
                    }

                    if (posPossibleErreur == -1) {

                        data[erreurs[cntr]] = somme / (key[2 + ii]);

                        inputData[erreurs[cntr]] <<= 8;

                        inputData[erreurs[cntr]] |= somme / key[2 + ii];

                        erreurs.erase(erreurs.begin() + cntr);

                        return;
                    }
                }
            }
        }
    }

    static void separer(int dim, uint64_t* inputData, unsigned char* key, int cntr, int dataLength, int
keyLength, std::vector<int>& erreurs, std::vector<int>& erreursHash) {

        int width = round(pow(exp(1.0), (std::log(keyLength) / dim)));
        int widthD = round(pow(exp(1.0), std::log(dataLength) / dim));

        // trouver le point:
        // effectuer les translations pour trouver un seul vrai, ce qui met la faute sur le hash

        for (int w = 1; w <= dim; w++) {

            for(int y = -1; y <= 1; y++) {

                for (int u = 1; u < width; u++) {

                    if (y != 0) {
```

```
                int indiceOppose = erreurs[cntr];

                int indice = 0;
                int i = y * u * pow(width, w) - 1;

                for (int k = dim; k >= 0; k--) {

                    if (pow(width, k) <= i) {

                        indice -= (int)(i / pow(width, k)) * pow(widthD, k);
                        i -= pow(width, k);
                    }
                }

                // effectuer la convolution:

                if (indiceOppose + indice < 0) indiceOppose = 0;
                else indiceOppose += indice;

                uint64_t somme = inputData[indiceOppose] >> 8;
                uint8_t hash = (uint8_t)((inputData[indiceOppose] << 56) >> 56);

                int posPossibleErreur = -1;

                for (int j = pow(width, dim) - 1; j >= 0; j--) {

                    int indice = indiceOppose;
                    int i = j;

                    for (int k = dim; k >= 0; k--) {

                        if (pow(width, k) <= i) {

                            indice += (int)(i / pow(width, k)) * pow(widthD, k);
                            i -= pow(width, k);
                        }
                    }

                    if (indice < dataLength) {

                        uint8_t valeur = (uint8_t)((inputData[indice] << 56) >> 56);

                        if (j == 0) somme /= (key[2 + j]);
                        else somme -= valeur * key[2 + j];

                        bool testR = true;

                        for (int f = 0; f < erreurs.size(); f++) {

                            if (indice == erreurs[f])
                            {
                                testR = false;
                            }
                        }

                        if (!testR) {

                            posPossibleErreur = j;
                            break;
                        }
```

```
                            }
                        }

                        if (hash == somme && posPossibleErreur == -1) {

                            erreursHash.push_back(erreurs[cntr]);

                            erreurs.erase(erreurs.begin() + cntr);

                            return;
                        }
                    }
                }
            }
        }
    }

    static void deleteErreurs(int dim, uint64_t* inputData, unsigned char* key, int cntr, int
dataLength, int keyLength, std::vector<int>& erreurs) {

        int width = round(pow(exp(1.0), (std::log(keyLength) / dim)));
        int widthD = round(pow(exp(1.0), std::log(dataLength) / dim));

        std::vector<int> liste;

        for (int j = pow(width, dim) - 1; j >= 0; j--) {

            int indice = cntr;
            int i = j;

            for (int k = dim; k >= 0; k--) {

                if (pow(width, k) <= i) {

                    indice += (int)(i / pow(width, k)) * pow(widthD, k);
                    i -= pow(width, k);
                }
            }

            if (indice < dataLength) {

                liste.push_back(indice);
            }
        }

        for (int i = 0; i < liste.size(); i++) {

            for (int j = 0; j < erreurs.size(); j++) {

                if(erreurs[j] == liste[i]) erreurs.erase(erreurs.begin() + j);
            }
        }
    }
};
```

BIBLIOGRAPHIE

(1)