Code correcteur d'erreurs

Projet

Thomas Lavaur

encadré par
Mr. Philippe Gaborit

Master 2 cryptis - 2020/2021

Table des matières

1 Algorithme ISD

1.1 Principe de l'algorithme

Le but de l'algorithme est de retrouver un mot du code à partir d'un syndrome avec des erreurs. C'est une des meilleurs attaques proposée contre les protocoles qui reposent sur la récupération de l'erreur à partir du syndrome. C'est donc une attaque potentielle contre le chiffrement MDPC de la section 2. L'algorithme ISD de Prange repose sur la supposition de pouvoir deviner k coordonnées (k étant la dimension du code) qui n'ont pas d'erreur afin de pouvoir reconstruire le mot du code. Pour cela on génère des permutations aléatoires jusqu'à ce que les $n-k$ premières coordonnées du nouveau syndrome obtenues aient un poids assez faible pour résoudre un système bien plus simple.

1.2 Fonctions du programme

1.2.1 Structure matrice et fonctions associées

Une matrice est simplement un tableau de valeurs entières avec ses dimensions. La fonction *val* permet de récupérer la valeur dans la matrice à une certaine position et *pt* renvoie le pointeur de cette position afin de pouvoir la modifier. Enfin, *destruction_matrice* permet juste de free l'espace alloué d'une matrice.

1.2.2 Fonction d'aléatoire

La fonction *rand* fournie par C ayant une période assez courte modulo 2, j'ai pris la décision de prendre chaque bit généré à chaque itération de *rand* afin de rallonger la période et de ne pas avoir de lignes linéairement dépendantes dans la génération des matrices. Tout cela est donné dans la fonction *bit_generator*.

1.2.3 Fonctions produit, transposition et inverse

Ces fonctions font ce qu'indique leur nom sans réelle spécificité. La fonction *produit_matrice* renvoie la première matrice d'entrée si les dimensions ne sont pas compatibles. La fonction *inverse_matrice* fonctionne par pivot de Gauss et suppose que la matrice d'entrée est inversible.

1.2.4 Fonction génératrice de permutation

La generation de permutation fonctionne de la manière suivante dans *generation_permutation* : dans un premier temps, on crée un tableau d'indice correspondant à la case i au numéro de la colonne ou se trouve de 1 dans la ligne i. On l'initialise donc avec la liste $(1, 2, 3, \ldots, n)$ puis on mélange, dans un second temps, de façon à avoir une probabilité uniforme d'obtenir chaque nombre dans chaque case. Enfin, on crée la matrice correspondante pour la renvoyer.

1.2.5 Fonction *gauss_jordan*

gauss_jordan fonctionne de la même manière que l'inversion décrite ci-dessus, c'est-à-dire par pivot de Gauss. Si l'échelonnage aboutit à l'impossibilité de faire apparaître la matrice identité sur la partie gauche, alors on renvoie une matrice vide codée par un nombre de ligne et de colonne valant 0. Sinon on renvoie la matrice qui permet d'obtenir cette forme, c'est-à-dire la matrice U telle que pour une matrice H donnée, $U = (I||H')$.

1.3 Temps de calcul d'ISD

Pour les paramètres $n = 400$; $k = 200$; $t = 20$; et sur 1000 essais, le temps moyen d'exécution d'ISD est de 1,308 secondes.

Pour les paramètres $n = 1000$; $k = 500$; $t = 10$; et sur 1000 essais, le temps moyen d'exécution d'ISD est de 1,616 secondes.

2 Chiffrement MDPC

2.1 Principe de l'algorithme $bitflip$

L'algorithme $bitflip$ consiste à retrouver un mot de code correspondant à l'entrée qui est un mot comportant une ou plusieurs erreurs. Cet algorithme est très simple et repose sur le fait de modifier les bits qui apparaissent dans la génération du plus d'erreur. Pour chaque bit, on compte le nombre d'opérations dans lesquelles il apparaît et qui résulte par un 1 dans le syndrome. On change ensuite tous les bits au-dessus d'un certain seuil en espérant ainsi diminuer le poids du nouveau syndrome. On répète cela jusqu'à ce que le syndrome soit 0 (si l'on souhaite un mot du code) ou qu'il soit inférieur à un certain poids.

2.2 Principe du protocole

La génération des clés correspond au choix d'une matrice génératrice d'un code MDPC quasi-cyclique. Pour cela, on n'a pas besoin de générer les matrices mais seulement leur première ligne car celles-ci sont circulantes.

On a donc la génération des clés h_0 et h_1 avec h_0 inversible qui correspond à la matrice génératrice du code et qui seront donc nos clés privées. Le produit $h_0^{-1}h_1 = h$ sera la clé publique.

On chiffre alors chaque bloc de la taille de la fonction de hashage choisie de la façon suivante : on tire aléatoirement deux erreurs e_0, e_1 dont la somme des poids est fixée. On a alors un secret que l'on pourra partager : le hashé de ces deux erreurs qui sert de masque à notre message. Le message crypté c_1 est alors le produit de h par e_1 additionné à e_0 afin que seul celui qui connaît h_0 et h_1 puisse retrouver l'erreur à l'aide de l'algorithme $bitflip$. La seconde partie du crypté c_0 est le message recouvert par le masque.

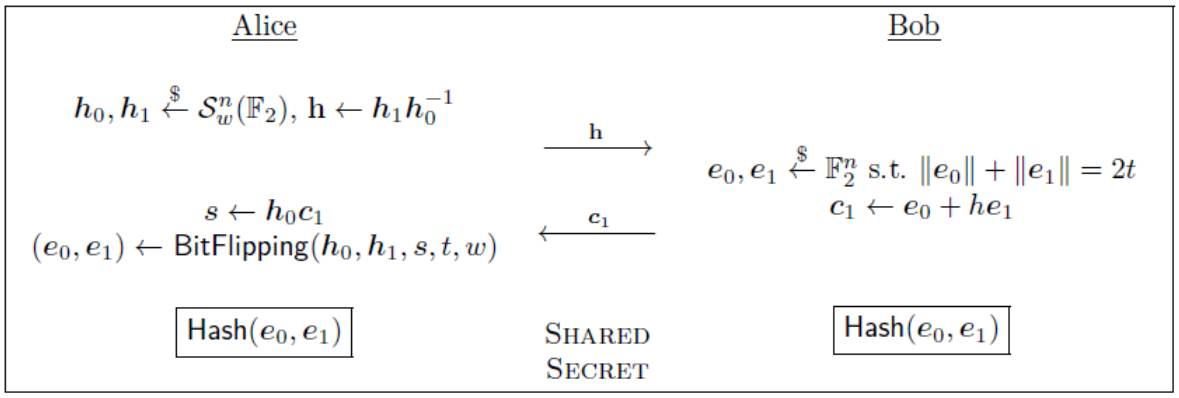
Pour déchiffrer, on calcul s le produit de c_1 par h_0. On obtient alors un syndrome bruité que l'on peut décoder avec l'algorithme $bitflip$ puisque l'on connaît la matrice du dual.

Cela fonctionne car :

$$h_0 c_1 = h_0(he_1 + e_0) = h_0 h e_1 + h_0 e_0 = h_0 h_0^{-1} h_1 e_1 + h_0 e_0 = h_0 e_0 + h_1 e_1 = s$$

Ce qui correspond exactement au syndrome avec erreur. Avec ce syndrome, on peut retrouver un mot du code et donc en déduire l'erreur qui permet de retrouver le secret partagé et de retirer le masque de chiffrement.

Rappel du schéma :



2.3 Fonctions du programme

2.3.1 Structures polynômes et fonctions associées

Contrairement à la section ISD on utilise ici une approche polynomiale ou de simple vecteur ligne. On a différentes structures qui ne sont la que pour aider la compréhension du programme mais qui peuvent toutes se résumer à des tableaux d'entiers. Pour faciliter le code, la fonction $val\_pol(P, i)$ renvoie la valeur du polynôme P au coefficient i, la fonction $val\_mat(P, i, j)$ renvoie la valeur du polynôme P qui correspond à la matrice circulante associées à la ligne i colonne j. Enfin, $pt(P, i)$ renvoie le pointeur associé pour pouvoir modifier le i-ème coefficient de P. La fonction du poids de Hamming fonctionne de façon classique.

2.3.2 Fonction de generation

La fonction de génération de polynôme repose sur le même principe que celle de l'exercice 1 pour la génération de permutation. On génère d'abord un polynôme où les premiers coefficients sont 1 et les autres 0 pour avoir le bon poids, puis on mélange pour un résultat aléatoire uniforme.

2.3.3 Fonctions multiplication et inversion

Pour la fonction *multiplication_polynome*, on remarque que la multiplication de deux matrices circulantes donne une nouvelle matrice circulante et donc on peut adapter la multiplication pour gagner du temps car cela correspond à calculer seulement la première ligne.

La fonction *inversion_polynome* est la seule fonction qui utilise une approche matricielle par pivot de Gauss car je n'ai pas trouvé d'autre moyen en C. Il serait sûrement beaucoup plus rapide de considérer une inversion par Euclide étendu mais il faudrait pour cela implémenter la division euclidienne dans $\mathbb{F}_2$.

2.3.4 Fonction *keygen*

On reprend exactement le protocole mais on génère un nouvel h_0 jusqu'à ce qu'il soit inversible. Puisque le polynômes doivent être aléatoire, de poids impaire et de longueur un nombre premier, h_0 a de très fortes probabilités d'être directement inversible et je ne suis jamais tomber sur un cas où il ne l'était pas.

2.3.5 Fonctions *hash_polynome* et XOR

La fonction XOR réalise le XOR coefficient par coefficient de chaque polynôme d'entrée mais de la longueur du premier. Celui-ci doit être plus petit que la longueur en octet de la fonction de hashage.

La fonction *hash_polynome* prend deux polynômes d'entrée (correspondant aux deux erreurs e_0 et e_1), les concatènes et renvoie dans un polynôme (dans $\mathbb{Z}$) la valeur décimale de chaque octet du hashé.

2.3.6 Fonction de chiffrement et de déchiffrement

Ces fonctions correspondent exactement à celles décrites dans le protocole.

2.4 Temps de calcul

2.4.1 Exécution standard

Algorithme	Keygen	Encryption	Decryption
Temps en s	45	0,02/bloc	1,2/bloc

2.4.2 Variations sur le seuil

Valeur de T	21	22	23	24	25	26	27	28	29
Temps du déchiffrement en s/bloc	∞	$1,54$	$1,2$	$1,2$	$1,2$	$1,2$	$1,5$	$1,8$	∞

2.4.3 Variations sur le poids

Valeur de w	39	41	43	45
Temps de Keygen en s	45	41	42	∞
Temps de chiffrement en s/bloc	$0,02$	$0,02$	$0,02$	∞
Temps du déchiffrement en s	$1,2$	$1,2$	$1,5$	∞