Compte-rendu

Réseau Euclidiens

@cypri3

2024

Table des matières

Introduction

Cette implémentation repose sur deux ouvrages principaux comme proposés :

Les chapitres 19.9 et 19.10 du livre de Steven Galbraith présentent, dans ses grandes lignes, le système GGH et les principales méthodes de cryptanalyse qui ont été trouvées pour ce système :

— Galbraith, S. D. *Mathematics of public key cryptography*, version 2.0, 2018, Part IV Lattices.

L'article originel de Goldreich, Goldwasser et Halevi, qui contient plus de détails :

— Goldreich, O., Goldwasser, S., & Halevi, S. (1997). *Public-key cryptosystems from lattice reduction problems*. In Advances in Cryptology—CRYPTO'97 : 17th Annual International Cryptology Conference Santa Barbara, California, USA August 17–21, 1997 Proceedings 17 (pp. 112–131). Springer Berlin Heidelberg.

Le programme est disponible en deux versions : une première en format Jupyter Notebook et une seconde en format Sage. Il se décompose en deux parties principales :

— **Tests de performances** (disponibles uniquement en format Notebook) : ces tests sont indépendants du projet principal. Ils incluent des comparaisons, des explications, et des résultats qui ont influencé l'implémentation finale du cryptosystème. Les tests permettent d'évaluer différentes optimisations et leur impact sur les performances.

— **Projet principal** : il contient toutes les fonctions nécessaires au cryptosystème et un ensemble de programmes de test. La version Notebook présente une structure organisée, comprenant successivement :

  * l'implémentation du cryptosystème ;
  * des tests unitaires pour valider les fonctionnalités ;
  * les attaques (Embedding, Nearest Plane, etc.) ;
  * un programme de test général combinant toutes ces parties.

  La version Sage est divisée en quatre fichiers distincts, qui doivent être utilisés dans un ordre précis :

  * `fonctions_GGH.sage` : contient l'implémentation principale du cryptosystème (clé publique, clé privée, chiffrement, déchiffrement, etc.) ;
  * `test_cryptosysteme.sage` : permet de tester les fonctions du cryptosystème de manière indépendante ;
  * `test_attaques.sage` : regroupe les implémentations des attaques pour évaluer leur efficacité ;
  * `test.sage` : une version synthétique combinant les tests du cryptosystème et des attaques pour une exécution complète.

Dans le cas où la version Notebook ne pourrait pas être exécutée, une version pré-exécutée sera jointe en PDF pour fournir une documentation complète des résultats obtenus. Si de nouveaux résultats sont nécessaires, il suffit d'exécuter les cellules du Notebook dans l'ordre proposé.

Notice d'utilisation

Pour utiliser l'implémentation, il est nécessaire d'avoir SageMath installé, en version Notebook ou console. Voici les instructions détaillées pour chaque version :

Version console

L'exécution des différents programmes de test via console nécessite de suivre ces étapes dans l'ordre :

1. Commencez par charger les fonctions principales en exécutant le fichier `fonctions_GGH.sage` :

   ```
   load("fonctions_GGH.sage")
   ```

2. Ensuite, lancez les programmes de test selon vos besoins :
   * Pour tester uniquement le cryptosystème :

     ```
     load("test_cryptosysteme.sage")
     ```

   * Pour évaluer les attaques :

     ```
     load("test_attaques.sage")
     ```

   * Pour effectuer un test complet (cryptosystème + attaques) :

     ```
     load("test.sage")
     ```

3. Chaque fichier de test contient des variables modifiables, dont le rôle est expliqué dans les commentaires situés au-dessus de chaque fonction. Ces variables, qui possèdent des valeurs par défaut, peuvent être adaptées selon les besoins pour personnaliser les tests, comme la taille des matrices ou les erreurs à générer. Pour une référence complète, une annexe détaillée est incluse à la fin du document. Elle répertorie toutes les variables disponibles, leurs utilisations, ainsi que leurs valeurs par défaut.

Version Notebook

La version Notebook est préconfigurée pour offrir une expérience utilisateur simplifiée. Les cellules sont déjà exécutées dans la version fournie, mais si de nouveaux résultats sont nécessaires :

1. Ouvrez le fichier Notebook dans Jupyter.
2. Exécutez chaque cellule dans l'ordre indiqué, depuis l'implémentation des fonctions jusqu'aux tests généraux.
3. Les sections sont clairement identifiées, et les résultats intermédiaires sont affichés directement après chaque exécution.

Description de l'implémentation

Le code implémente :

— **Génération des clés** (`KeyGen`) : création des matrices privées et publiques.

— **Chiffrement** (`Cipher`) : ajout d'une erreur au message clair avant chiffrement.

— **Déchiffrement** (`Decipher`) : reconstruction du message clair.

— **Chiffrement/déchiffrement de messages** : transformation des messages en binaire et gestion de padding basique.

— **Trois attaques** : brute force sur l'erreur, nearest plane attack, et embedding attack.

— **Tests unitaires et globaux** : validation des différentes étapes du système.

Nous allons détaillers les diverses spcificitées de l'implémentation de ces différentes fonctions.

Génération de clés publiques et privées

La première étape consiste à générer la clé privée. Pour cela, des nombres aléatoires sont tirés uniformément entre -4 et 4, conformément à la proposition de Goldreich, Goldwasser et Halevi. Ces petits coefficients rendent la clé efficace pour certaines opérations. Ce paramètre peut cependant être ajusté selon les besoins.

Ensuite, pour garantir l'inversibilité de la matrice publique, il est nécessaire que la matrice privée ait un déterminant différent de zéro. Une vérification a donc été effectuée en ce sens. Cependant, une difficulté notable a été de comprendre l'importance d'ajouter une matrice identité multipliée par k, avec $k = \text{round}(\sqrt{n}) \times 4$. Cette étape, peu mentionnée dans les propositions d'implémentation, s'est avérée essentielle après de nombreuses recherches et tests. Sans elle, la plupart des messages deviennent indéchiffrables, sauf dans le cas des matrices unimodulaires triviales.

Trois approches principales ont été testées pour la génération de matrices unimodulaires :

— Application d'opérations élémentaires sur des matrices identité.

— Multiplication de matrices triangulaires.

— Implémentation proposée par Goldreich, Goldwasser et Halevi.

Résultats des tests de performances

Les tests de performances visent à évaluer l'efficacité des différentes méthodes employées dans le projet. Le premier test porte sur la génération de matrices triangulaires supérieures et inférieures de manière optimale.

Pour ce test, il a été constaté qu'il était plus efficace de générer une matrice complète de taille $n \times n$ avec des coefficients aléatoires, puis de remplacer les coefficients indésirables par des 0 ou des 1 pour obtenir une matrice triangulaire de la bonne forme. Cette approche, bien que moins intuitive, s'est révélée plus rapide que la génération du nombre exact de coefficients requis, notamment lorsque cette dernière était effectuée à l'unité avec la bibliothèque `random`. L'efficacité de cette méthode repose sur l'exploitation de codes compilés en C, qui gèrent mieux les ressources et optimisent les boucles de génération des nombres aléatoires.

Une alternative intéressante aurait pu être de générer directement le nombre exact de coefficients nécessaires dans une matrice de taille $(n/2) \times n$ avant de les placer dans une matrice triangulaire finale. Cependant, cette méthode n'a pas été explorée dans ce travail.

La comparaison entre la méthode triangulaire et celle basée sur les opérations élémentaires sur une matrice identité a révélé que la méthode triangulaire était considérablement plus rapide. Cependant, elle présentait un biais sur l'un des coefficients. La solution trouvée a été de combiner les deux méthodes : utiliser la multiplication de matrices triangulaires tout en appliquant des opérations élémentaires pour corriger ce biais. Cette approche s'est avérée bien plus efficace, notamment pour des matrices de grande dimension. Les résultats sont visibles dans le fichier de test de performance. Toutefois, comme indiqué, le choix final a été de rester sur l'implémentation proposée dans les papiers, malgré les gains de performance possibles avec cette optimisation.

Dans certains cas, il a été observé que les tableaux en Python offraient des performances supérieures à celles des objets `matrix` de Sage. Cette différence semble résulter de la complexité des objets du langage, tandis que les opérations élémentaires sur des tableaux Python natifs sont mieux optimisées. Ce constat met en lumière l'importance de tester diverses solutions, même au sein d'un même langage, afin de déterminer celles offrant le meilleur compromis entre simplicité d'implémentation et performances. Il est essentiel de prendre le temps d'effectuer ces tests avant de se tourner systématiquement vers des bibliothèques externes, car elles ne garantissent pas toujours les meilleures performances.

Une optimisation majeure dans l'implémentation des matrices unimodulaires a consisté à utiliser `matrix.identity(n)` plutôt que `identity_matrix(ZZ, n, n)`, permettant un gain d'un facteur 50 pour des matrices de taille 200. Ces chiffres ne sont pas directement présents dans les tests de performance, mais peuvent être obtenus en remplaçant la ligne correspondante dans la fonction de génération de matrice unimodulaire de l'implémentation du cryptosystème.

Fonction de chiffrement

Pour le chiffrement, des erreurs de norme relativement petite sont générées, avec des valeurs entre -1 et 1. Ces erreurs sont conçues pour faciliter les attaques et rendre le déchiffrement moins sujet à des erreurs. Les vecteurs d'erreurs sont composés de nombreux zéros, réduisant ainsi les risques de déchiffrement incorrect tout en simplifiant les attaques par force brute.

Pour limiter le temps de recherche de ces vecteurs, un *breakpoint* est utilisé : au-delà d'un certain nombre d'itérations, une méthode systématique est employée pour construire l'erreur.

Fonction de déchiffrement

Le déchiffrement suit une approche classique, mais inclut deux méthodes d'arrondi différentes. En Python, l'arrondi des nombres négatifs suit la logique des entiers non signés, par exemple `-0.5` est arrondi à `-1`. Une méthode alternative, arrondissant à 0, est aussi présente sous forme de commentaire dans le code.

Chiffrement et déchiffrement de messages

Un système complet de chiffrement et déchiffrement a été conçu et implémenté. Les messages sont convertis en chaînes binaires, découpés en blocs de taille multiple de 8, et organisés en vecteurs. Ce choix repose sur la simplicité qu'offre la représentation binaire, mais également sur l'efficacité obtenue en utilisant des matrices de taille modérée, faciles à générer et à manipuler.

Bien que le système encode les messages uniquement avec des vecteurs de 0 et 1, le cryptosystème utilisé aurait permis une représentation plus riche, en exploitant des vecteurs appartenant à $\mathbb{Z}^n$ avec des coefficients modérés. Une telle approche aurait été plus avantageuse d'un point de vue théorique, notamment en termes de théorie de l'information, en permettant une densité d'encodage supérieure. Cependant, cette solution se heurtait à des difficultés pratiques, notamment pour gérer des matrices de tailles variables et des implémentations adaptées aux messages de longueurs arbitraires.

L'utilisation d'un *padding* simple, composé uniquement de zéros, a été adoptée pour compléter les blocs de taille fixe. Ce choix, bien que vulnérable à certaines attaques cryptographiques, a été retenu pour sa simplicité et son adéquation avec les objectifs éducatifs du projet.

Implémentation des attaques

Trois attaques principales ont été réalisées dans le cadre de ce projet :

— Une attaque par force brute sur l'erreur.

— Une *Nearest Plan Attack*.

— Une *Embedding Attack*.

Force brute sur l'erreur

Cette attaque consiste à générer des vecteurs d'erreur candidats et à tester leur validité jusqu'à ce qu'une solution soit trouvée. Bien que le concept soit théoriquement simple, l'optimisation de cette dernière s'est avérée essentielle pour limiter la consommation de mémoire. Une implémentation naïve nécessitait plus de 32 Go de RAM pour des matrices de taille 200, ce qui a conduit à l'utilisation de la fonction `yield`. Elle permet d'itérer en partie sur des fonctions compilées en C. Cela permet non seulement d'accélérer l'exécution du programme, mais aussi de limiter l'usage de la mémoire en traitant chaque vecteur au fur et à mesure de sa génération, plutôt que de les construire tous en mémoire.

Pour limiter le temps d'exécution, l'attaque s'arrête dès qu'une bonne solution est trouvée. Cependant, toutes les solutions potentielles découvertes avant celle-ci sont également retournées, permettant ainsi de détecter d'éventuels faux positifs. Bien que cette approche soit rudimentaire, elle s'est révélée efficace pour les matrices de petite taille et parfois même pour des matrices plus grandes.

Nearest Plan Attack

L'attaque par *Nearest-Plane* repose sur la réduction de réseau et consiste à rechercher l'erreur de chiffrement sous forme de combinaison linéaire des vecteurs de la base de la matrice publique. La principale difficulté réside dans la compréhension mathématique de l'attaque plutôt que dans son implémentation.

L'implémentation de cette dernière débute par la réduction de la matrice publique B_{pub} à l'aide de l'algorithme LLL, puis l'on applique l'inverse de cette matrice réduite au message chiffré c. L'erreur est alors approximée en arrondissant les éléments du vecteur résultant. Ensuite, un processus itératif de correction est appliqué, où chaque itération ajuste le message chiffré en fonction de l'erreur estimée. À la fin de ce processus, une approximation du message clair est obtenue.

Dans notre situation, les résultats sont très satisfaisants. En effet, la simplicité des erreurs utilisées permet de retrouver le message clair dans la grande majorité des cas.

Embedding Attack

L'attaque *embedding* repose sur la construction d'une matrice augmentée qui intègre à la fois la matrice publique B_{pub} utilisée pour le chiffrement et le message chiffré c. Une ligne supplémentaire, appelée "ligne d'ancrage", est ajoutée à cette matrice. Cette ligne est conçue de manière à faciliter l'utilisation de l'algorithme LLL (Lenstra–Lenstra–Lovász) pour réduire la matrice et isoler l'erreur de chiffrement. Une fois l'erreur e obtenue à partir de cette réduction, une transformation linéaire permet de calculer une approximation du message clair en soustrayant l'erreur du message chiffré, puis en appliquant l'inverse de la matrice publique.

La principale difficulté de cette attaque résidait également dans sa compréhension mathématique. Les résultats obtenus sont excellents pour des erreurs composées de petites valeurs, comme -1 et 1, mais l'efficacité diminue significativement lorsque l'erreur comprend d'autres valeurs, telles que 3 et -3, limitant ainsi les cas où cette méthode peut être appliquée avec succès.

Tests et performances des attaques

Les tests des attaques ont été réalisés sur des matrices de tailles variées, allant de 16 à 200, afin d'assurer une évaluation exhaustive tout en limitant le temps d'exécution. Certains paramètres ont volontairement été choisis à des valeurs relativement faibles pour permettre une bonne réalisation ces dernières, le tout en un temps raisonnable et en maintenant la pertinence des résultats. Il est recommandé de ne pas activer le mode débogage de la fonction `keygen()` pour les grandes tailles de matrice, afin d'éviter des ralentissements inutiles.

Programme de test

Plusieurs tests unitaires sont intégrés à l'implémentation pour vérifier individuellement les fonctionnalités du code. Par exemple, des tests s'assurent que pour des messages et matrices générés aléatoirement, le chiffrement et le déchiffrement fonctionnent comme prévu. Les tests de performance, bien qu'importants, ne sont pas intégrés au programme principal afin de ne pas alourdir le code ni ralentir l'exécution sur un grand nombre de matrices. Le programme de test principal effectue :

— Une génération de paire de clés de taille 16, avec affichage.

— Le chiffrement et déchiffrement d'une phrase, avec affichage du texte chiffré.

— L'exécution des trois attaques implémentées.

Les paramètres de ces tests ont été choisis pour obtenir des résultats lisibles et rapides à exécuter, avec un affichage optimisé pour une utilisation en console. Ils peuvent cependant être ajustés pour évaluer les performances sur des matrices de plus grande taille. Par exemple, le temps d'exécution du test global pour des matrices de taille 200 est d'environ 2 minutes 20 sur un ordinateur standard.

Conclusion

L'implémentation d'un système de chiffrement basé sur le cryptosystème GGH a permis une mise en pratique concrète de concepts essentiels en cryptographie sur les réseaux. Chaque étape, de la génération des clés à la réalisation des attaques, a été conçue pour trouver un équilibre entre simplicité et optimisation. Bien que ce cryptosystème soit aujourd'hui considéré comme obsolète, le travail de recherche, de compréhension et d'amélioration réalisé s'est avéré particulièrement enrichissant.

Les tests ont montré que le système fonctionne correctement pour des dimensions raisonnables, tout en révélant des limites pour des tailles plus importantes, qui nécessiteraient des techniques plus avancées. Des pistes d'amélioration existent, notamment l'exploration de représentations alternatives pour les messages ou la mise en place d'autres optimisations algorithmiques.

Travailler sur un cryptosystème exploitant des erreurs, un principe contre-intuitif dans un contexte de sécurité, a également été une expérience très formatrice. Ce projet constitue une base solide pour mieux comprendre les cryptosystèmes basés sur les réseaux, en particulier leurs variantes post-quantiques, comme Kyber ou Dilithium.

Annexe : Principales Notations et Variables

Cette annexe présente les principales notations et variables utilisées dans le programme principal. Ces notations sont également respectées dans la majorité des tests, mais peuvent légèrement varier dans le programme d'évaluation des performances. Chaque variable est accompagnée de sa définition, son rôle dans le programme, et sa valeur par défaut ou suggérée, lorsqu'elle est pertinente.

- n (`int`) : Taille de la matrice carrée $n \times n$. Définit la dimension des matrices générées dans le cryptosystème.
- `nbc` (`int`) : Nombre de matrices unimodulaires multipliées entre elles pour générer la matrice publique. Une valeur suggérée est $2n$.
- l (`int`) : Amplitude des coefficients dans la matrice privée. Valeur par défaut : 4.
- `debug` (`bool`) : Si activé (`True`), affiche les matrices générées pour le débogage et les vérifications intermédiaires.
- `Bpriv` (`Matrix`) : Matrice privée utilisée pour le chiffrement et le déchiffrement. Générée de manière aléatoire selon les paramètres spécifiés.
- `Bpub` (`Matrix`) : Matrice publique obtenue en multipliant `Bpriv` par des matrices unimodulaires (`U`).
- `U` (`Matrix`) : Matrice unimodulaire utilisée pour transformer `Bpriv` en `Bpub`.
- `m` (`vector`) : Message clair, représenté sous la forme d'un vecteur $m \in \mathbb{Z}^n$.
- `breakPoint` (`int`) : Nombre maximal d'itérations pour ajuster l'erreur dans le cadre d'une attaque. Valeur par défaut : 128.
- `c` (`vector`) : Message chiffré résultant du produit de `Bpub` par le message m, auquel est ajouté un vecteur d'erreur.
- `nb` (`int`) : Nombre de tests à effectuer dans les expérimentations ou dans les attaques.
- `size` (`int`) : Taille des blocs pour le chiffrement et le déchiffrement de messages sous forme de chaînes de caractères. Doit être un multiple de 8. Valeur par défaut : 112.
- r (`int`) : Borne supérieure pour la norme Euclidienne des vecteurs candidats, utilisée dans l'attaque par force brute.
- `borneInf` (`int`) : Borne inférieure des coefficients des vecteurs candidats pour l'attaque. Valeur par défaut : 0.
- `borneSup` (`int`) : Borne supérieure exclusive des coefficients des vecteurs candidats pour l'attaque. Valeur par défaut : 2. Cela signifie que l'espace des coefficients de m est défini comme [`borneInf`, `borneSup`[.