Mathématiques en technologies de l'information - Travail pratique

Craquer un message RSA

Antoine Sutter
HEPIA - premier semestre

Introduction

Dans ce travail pratique de mathématiques en technologies de l'information, le but est de réussir à lire un message chiffré à l'aide de l'algorithme RSA. Le RSA (acronyme de Rivest, Shamir et Adleman, les noms des créateurs) est un algorithme de chiffrement dit asymétrique. Cela signifie qu'afin de pouvoir transférer un message, deux clés sont nécessaire: une clé pour chiffrer le message et une autre clé pour déchiffrer le message. Ces clés sont généralement appelées clé publique et clé privée respectivement. Ce fonctionnement diffère d'un algorithme dit symétrique ou la même clé est utilisée pour le chiffrement et le déchiffrement, comme dans l'algorithme de César.

Cependant, le but de ce travail pratique n'est pas de simplement lire un message chiffré à l'aide de l'algorithme RSA en utilisant une clé publique et une clé privée mais d'y parvenir uniquement à l'aide de la clé publique. Le vrai défi est donc parvenir à calculer la clé privé.

Utilisation du RSA

Imagine que deux personnes veulent s'envoyer des message chiffrés, Alice et Bob. Voici donc les étapes afin que ceci puisse se produire en utilisant l'algorithme RSA:

1. Alice génère une pair de clés publique et privé
2. Bob génère une pair de clés publique et privé
3. Alice écrit son message
4. Alice demande la clé publique de Bob
5. Alice chiffre son message à l'aide de la clé de Bob
6. Alice envoie son message chiffré à Bob
7. Bob déchiffre le message d'Alice en utilisant sa clé privé

But du travail pratique

Dans l'algorithme RSA, une clé publique consiste en 2 nombres, traditionnellement appelés **n** et **e**. Un des principe mathématique assurant la sécurité de l'algorithme repose sur la manière dont le nombre **n** a été généré. En effet, celui-ci est le résultat de la multiplication de 2 nombres premiers. Par construction, il est **n** et donc uniquement divisible par ces derniers (ainsi que 1 et **n** évidement mais ceux-ci ne nous avancent pas dans notre problème).

Ainsi, quand nous disons vouloir craquer le RSA, ce qui nous essayons concrètement de faire est factoriser **n**.

Choix techniques

Ayant l'habitude de travail avec Python dès qu'un projet est en rapport de près ou de loin avec les mathématiques, c'est avec ce langage que j'ai originellement commencé à travailler. Je ne suis néanmoins pas particulièrement familier avec ce dernier. Ainsi, suite à des problèmes de performances et d'utilisation mémoire, je me suis finalement tourné vers le C, un langage que je connais mieux et avec lequel j'ai donc plus de contrôle.

Le travail rendu est donc un projet C accompagné d'une configuration Makefile. Le projet a été réalisé à l'aide de Visual Studio Code sous Mac OS.

Cheminement

Comme expliqué au début du document, le problème principal à résoudre afin de réussir à craquer un message RSA sans posséder la clé privé associé et de réussir à factoriser un grand nombre en sa décomposition de deux nombres premiers. Plus précisément, il s'agit d'un des deux nombre qui forment à eux deux la clé publique, le nombre traditionnellement appelé **n**. C'est donc sur cette partie que je me suis concentré en premier.

Méthode génération de nombres premiers en Python

La première méthode qu'il m'a paru naturel de mettre en place est de d'abord générer une liste de nombres premiers entre **0** et la racine carrée de **n** afin d'ensuite tester si un de ces nombres premier divise **n**. Pour ceci, je me suis tourné vers l'algorithme Sieve of Eratosthenes[1]. C'est en tentant d'implémenter cette algorithme en Python que je me suis heurté aux problèmes de performances et d'utilisation mémoire que j'ai cité dans la section "Choix technique". En effet, cet algorithme fonctionne de la manière suivante:

1. Créer une liste de nombres allant de 2 à la racine carrée de **n**
2. Partir du principe que tous ces nombres sont premiers
3. Commencer au début de la liste et vérifier si le nombre actuel est premier
4. Si oui, parcourir toute la site pour mettre à jour les multiples de ce nombre
5. Passer au nombre suivant

Après avoir parcouru toute la liste, les seules valeurs restantes seront les nombres premiers entre 2 et la racine carrée de **n**. Malheureusement, la complexité de cet algorithme est de taille $\mathbf{0(n^2)}$ et prends donc beaucoup de temps quand **n** grandit. Il est aussi inefficace en taille car il demande de garder en mémoire une liste contenant tous les nombres entre 2 et la racine carrée de **n**, même ceux qui ne sont pas premiers.

Ainsi, lors de l'implémentation de cet algorithme en Python, l'utilisation du processeur de ma machine atteignait immédiatement les 100% d'utilisation. Quant à la mémoire, celle-ci dépassait

[1] https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

les 30 Go, ralentissant de ce fait encore plus l'exécution du programme, jusqu'à le rendre complètement inutilisable.

Méthode génération de nombres premiers en C

Après avoir recommencé le projet cette fois-ci utilisant le langage C, j'ai décidé de remplacer la liste de nombres entre 2 et la racine carrée de **n** par une liste de nombre entiers non signés sur 64 bits et d'utiliser les bits individuel comme des valeurs booléen afin de déterminer si un nombre est premier. Ainsi, afin de savoir si, par exemple, le nombre 7 est premier, il suffit de regarder la valeur du bit équivalent à la valeur 2^7 dans le premier entier 64 bits.

Voici un exemple sur 8 bits. La premier ligne est la représentation binaire du byte. La deuxième ligne représente la valeur en puissance de 2 de chaque bit. La troisième ligne indique quel nombre correspond à quel bit.

0	1	0	1	0	0	0	0
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
7	6	5	4	3	2	1	0

Cette solution résout le problème d'utilisation de mémoire qui était présent sur le programme Python et accélère ainsi drastiquement le temps d'exécution du programme. Une fois la liste de nombre premiers générée, il suffit de la parcourir et tester si chaque nombre premier divise **n**.

Méthode brute force et brute force impair

Après avoir implémenté la méthode génération de nombres premiers, j'ai ensuite voulu implémenter une méthode de brute force afin de comparer le temps d'exécution. Cette méthode part de la racine carrée de **n** et tests tous les nombres en décrément jusqu'à 2. J'ai également implémenté une méthode brute force qui ne teste que les nombre pair.

Générer la clé privé

Une fois la factorisation de n terminée, il nous est maintenant possible de générer la clé privé afin de pouvoir déchiffrer le message. Avec n factorisé en deux nombres premiers appelés p et q, nous pouvons ensuite générer un nouveau nombre appelé l'indice d'Euler[2] (habituellement représenté avec la lettre ϕ) de la manière suivante :

$$\phi = (p - 1) * (p - 1)$$

Ensuite, nous pouvons utiliser ce nombre en combinaison avec **e** afin de trouver les coefficient de Bézout entre l'indice d'Euler et **e** (rappelle, **e** est le deuxième nombre avec **n** qui constitue la clé publique utilisé pour chiffrer le message).

[2] https://fr.wikipedia.org/wiki/Indicatrice_d%27Euler

Le théorème de bézout permet de trouver le plus grand dénominateur commun entre ϕ et **e**, ainsi que les coefficients **d** et **f** tels que :

$$(\phi * d) + (e * f) = pgdc(\phi, e)$$

Dans notre cas, nous somme intéressé par la valeur de **d** car celui-ci est en réalité la clé privé qui pourra ensuite être utilisé pour déchiffrer le message. Note que dans le théorème de Bézout, il est possible que la valeur de **d** soit négative. Dans le cas du RSA, si c'est le cas, il faut donc prendre la valeur suivante pour **d** :

$$d = d \bmod \phi$$

Avec cette étape fini, nous avons donc restitué la clé privé à partir de la clé publique et il est donc maintenant possible de déchiffrer le message.

Déchiffrer le message

La méthode utilisée pour déchiffrer le message est mathématiquement simple :

$$déchiffré = (chiffré \hat{} d) \bmod n$$

Cependant, lorsque **d** est trop grand comme dans notre cas ici, nous devons utiliser l'algorithme d'exponentiation rapide[3] afin de rendre ce calcul possible sur une machine 64 bits.

Afficher le message

En C, il est très simple d'afficher le message à l'aide de la fonction `printf` :

```
for (int i = 0; i < groupsLength; i++)
   printf("%s", (char *)&decoded[i]);
```

[3] https://fr.wikipedia.org/wiki/Exponentiation_rapide

Résultats et benchmarks

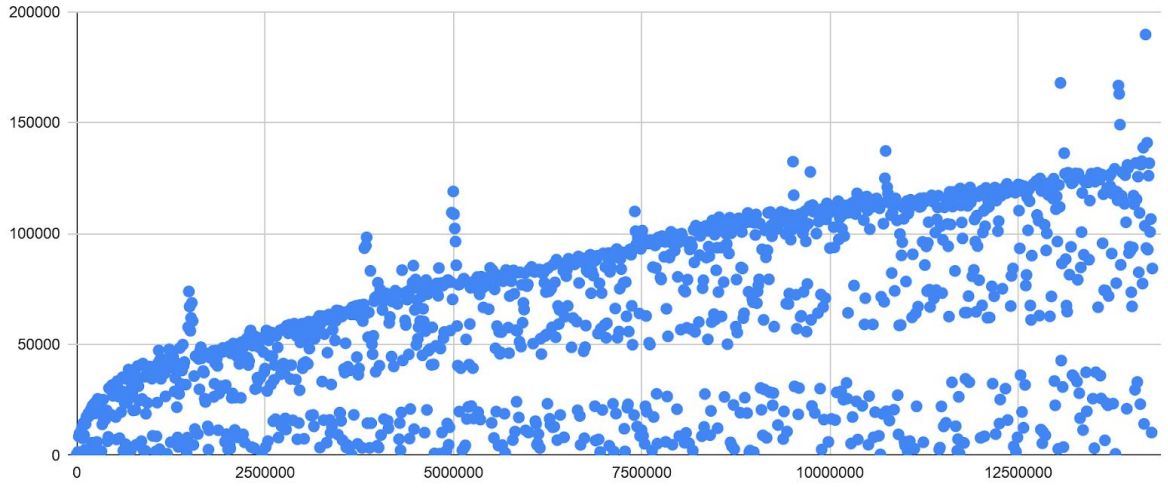
Le but du travail a donc correctement été remplis et le message a été déchiffré ! Une fois le programme lancé, voici le message affiché :

> Je suis connu pour un postulat non démontré. Mon réultat le moins célèbre est toutefois la pierre angulaire te permettant de déchiffrer ce message. Qui suis-je ?

Malheureusement, étant informaticien et non détective, je n'ai aucune idée de la signification de ce message. Quant à la performance, il s'agit de la partie qui m'a le plus surpris ! En effet, voici les résultat de performance des trois méthodes cités ci-dessus - génération de nombres premier, brute force et brute force impair respectivement :

```
Total time taken by CPU: 834µs
Total time taken by CPU: 88µs
Total time taken by CPU: 55µs
```

Les méthodes de brute force sont donc clairement beaucoup plus efficace. J'ai ensuite également écrit un petit programme afin de faire un test de performance sur la méthode brute force impair spécifiquement afin de voir son évolution comparé à **n** :



Axe vertical : le temps écoulé en tour d'horloge de CPU
Axe horizontal : **n**

Conclusion

Le message a correctement été déchiffré dans ce travail pratique. Cependant, ceci a uniquement été possible car le niveau de sécurité choisi par Dr. Eggenberg est volontairement faible. Dans le monde réel, il est simplement impensable de réussir une telle prouesse et ce travail m'a permis d'apprécier d'autant plus la beauté mathématique de cette algorithme et de son utilisation vaste et efficace depuis des décennies.

Sources

- https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes
- https://fr.wikipedia.org/wiki/Chiffrement_RSA
- https://fr.wikipedia.org/wiki/Indicatrice_d%27Euler
- https://fr.wikipedia.org/wiki/Th%C3%A9or%C3%A8me_de_Bachet-B%C3%A9zout
- https://fr.wikipedia.org/wiki/Exponentiation_rapide