Cryptanalyse

I] Vername Faible

1- Question1.py :

- Erreur d'exécution , dans certains cas le programme lève l'exception suivante

«« Traceback (most recent call last):
File "question1.py", line 112, in <module>
main()
File "question1.py", line 105, in main
print_doubleByte("y", cyphered)
File "question1.py", line 5, in print_doubleByte
print(varName + str(i) + " : " + value[i] + " = " + db_to_chr(value [i]))
UnicodeEncodeError: 'utf-8' codec can't encode character '\udab0' in position 24: surrogates not allowed
»»

une exception facilement détectable en tapant (for var in `seq 100 ` ; do python3 question1.py < test ; sleep 1 ; done) dans un shell tel que test est fichier teste qui contient une chaîne de caractères
L'erreur en raison de encodage de certains caractères utf-16 avec un encodage de Utf-8

- Génération de clé ne satisfait pas la contrainte sur la taille exigé dans l'exercice , pour pour rappelle " On utilise des clés de 8 octets qu'on décompose en 4 groupes de 2 octets k0k1k2k3. On suppose qu'on représente un caractère sur 2 octets "

c'est à dire soit on génère des clés de 2 Bytes pour chaque caractère soit on choisit un caractère asccii est on suppose qui il satisfait la condition (comme indiqué dans l'énoncé)

Or dans l'implémentation de de la fonction G dans le fichier Question1.py Génère des clés de 16 octets de '0 | 1' comme caractère ce qui dépasse largement les 8 bytes pour chaque clé

- il n'est pas demandé que chaque caractère du texte claire est sur 16 ce qui la fonction to_string fait dans la fonction to_doubleByt => une grande complexité en mémoire

2 - Question3.py :

- Utilisation de la modularité aurait aidé à organiser le travail, car le lancement du programme exécute tout d'abord le programme présent dans le fichier question1.py ce qui aurait pu être éviter avec utilisation d'un module qui accumule toutes les fonctions et l'importer dans le fichier question3.py et question1.py

- Pour rappelle " Mettez en œuvre votre méthode de cryptoanalyse et déterminez de façon expérimentale la longueur minimale d'un texte qui permet à votre programme de mener à bien l'analyse avec une probabilité au moins 1/2

”

c’est à dire étant donné un texte chiffré selon le système de exo1, le programme doit trouvé avec une probabilité de 1/2 la clé de chiffrement , Or le Programme de la question3.py ne respect pas cette règle car il prends en argument un texte (claire) et renvoie des résultat après “ le chiffrement ” ce qui le rends les teste très difficile car on pas accès au texte chiffré

- Concernant l’algorithme en commentaire on remarque quelque failles de sécurité <<< sous les même hypothèse → pas de caractère spéciaux dans le texte claire >>> tout d’abord E n’est pas toujours le caractère le plus répété dans n’importe quel texte (anglais / français) le mieux c’est de étendre cette liste jusqu’à 8 chars afin d’augmenter la probabilité de d’obtention de la clé, pour diminuer la complexité de algorithme il faut vérifier que pour chaque k[i] obtenu n’est on d’ hors du ascci si c’est le cas c’est à dire la lettre que on a supposé au début n’est pas correcte et il faut choisir un autre

pour la Méthode améliorer c’est une bonne approche mais la probabilité d’avoir toujours un E comme char le plus répété dans le chaque bloc de text[i%4] n’est pas très élevé, il existe la méthode de calcule de plausibilité qui nécessite pas des hypothèse, mais il est un peu difficile à mettre en ouvre (pas demander)

≡==

Conclusion:

- respecter les contraintes exigé dans l’énoncé (taille des clé, ce qui la question demande de programmer exactement)
- Utilisation de modularité python pour éviter les inclusion des fonctions main dans des autres
- corriger les erreurs d’exécution (on faisant attention aux encodage utiliser)

===

I] RSA

1-MR.py : < miller rabin primality >

A=970414662572987036738073380930068277976703888091620820472542132243449284026583907625475229569814127495415874295665009309485390291152974483670208651968070769578877807314678422841864995334772168343334091

B=661267326096401171386969593934715923507015891150925400534862594642388843193844643712827620046180758333

- Pour les nombres A , B le teste de miller rabin renvoie False Or A, B Sont des nombre Premiers (Erreur d’implémentation)

- Rabin-miller Primality test : est un teste Probabiliste qui a besoin d’un nombre d’itérations (souvent appelé witness de miller) ce nombre détermine La probabilité de fiabilité , => witness est proportionnel au nombre à tester (plus le nombre à tester est grand plus le witness doit être grand) Or dans le calcule de Probabilité mit en commentaire ne dépend pas de la taille nombre et le nombre de itérations = `floor(4 - log(100 - likelihood, 4))` tel que likelihood = 99

- Modification de (likelihood , >) génère des exceptions de type (math domain error) Or

pour les grand nombre et pour assurer une grand fiabilité il préférable de faire plusieurs tests > 100

2- RSA.py <chiffrement et déchiffrement >

- Utilisation de implémentation miller-rabin primalité test du fichier MR.py avec ces anomalies (taille , testes)

- Erreur d'exécution

changer le message de «hello» → «hello are you okey? » Génère L'exception suivante

««

```
Traceback (most recent call last): File "/tmp/sessions/abe571d70e93ef5b/main.py", line
152, in <module> main() File "/tmp/sessions/abe571d70e93ef5b/main.py", line 148, in main
print(decrypt_RSA(m, k[1])) File "/tmp/sessions/abe571d70e93ef5b/main.py", line 130, in
decrypt_RSA return mod.to_bytes(64, byteorder="little").decode("utf-8")
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xaf in position 1: invalid start byte
```

»»

En raison de encodage utiliser → impossible de chiffrer un texte plus grand

Le reste de l'implémentation de RSA est correcte En revanche certains failles de sécurité peuvent s'avérer très dangereuse

- Les Clés générer sont très petites est c'est dangereux car un Algorithme un peu solide peut facilement factoriser la clé publique pour en déduire la clé privé *** dans un temps raisonnable ***

- Il faut éviter de choisir e (exposant de chiffrement avec une manière bien définie) le mieux c'est de le choisir totalement au hasard car il existe des méthodes d'attaque de RSA développé par les organisme de Sécurité comme (NSA) qui est basé sur la manière dont on choisi E <source Wikipédia RSA et attaque >

Remarque : (Utile pour le chiffrement des grands messages)

Si On souhaite crypter des grands Messages la fonction `encrypt_RSA(msg, key)` factorise directement le message reçu en paramètre avec `int.from_bytes(bytes(msg, 'UTF-8'), byteorder="little")`

En revanche si le message est trop grands on risque de déborder la mémoire , découper en blocs aurait son efficacité d'optimisation

2- hybrid.py

- utilisation de clé RSA généré avec les fonctions du fichier RSA.py peuvent être erroné,
- utilisation de os.urandom(16) au lieu randByte() car c'est plus aléatoire , erreur d'exécution avec ma version (malgré que il es mise à jour)python de « cannot import name 'randbytes' from 'random' (/usr/lib/python3.8/random.py) » il fallait utiliser un interpréteur enligne

l’algorithme de factorisation de Pollard utilise en principe une la fonction $x \rightarrow x*x+1$
il n’a ne pas factoriser des grands nombre exemple (196216688000579533777013316
702486625249590164013243863054980091852402843979730131543580032282186078343)
qui est un nombre premier !
«« Le chiffrement et le Déchiffrement avec AES et RSA est correcte) »»

≡==

Conclusion

- il faut changer implémentation de algorithme miller-rabin primality test , car pour des grands (ou il devrait être utile) ne fonctionne pas correctement par conséquent la génération des clé RSA peut être erroné

- traitement des erreurs en raisons de encodage

- Les teste fournit sont top court il faut envisager une utilisation plus générale qu’un simple (beaucoup plus difficile à mettre en ouvre)

===

GOOD WORK