Analyse de Sécurité du Protocole de Génération de Secret par Puzzles Cryptographiques

https://github.com/yvan-allioux/secret-generation-with-cryptographic-puzzles

Rapport de TP

Yvan Allioux
Abir Hsaine

IAI 3 - 2023

lancement des scripts

Choix du programme à lancer dans le Dockerfile

```
# Lance un shell Bash à l'exécution du conteneur
#CMD ["/bin/bash"]
#CMD ["python", "main.py", "1000"]
CMD ["python", "modification.py", "1000"]
#CMD ["./auto.sh"]
```

#CMD ["/bin/bash"]
ouvre un shell dans le conteneur
#CMD ["python", "main.py", "1000"]
lance main.py avec un n de 1000
CMD ["python", "modification.py", "1000"]
lance modification.py avec un n de 1000
#CMD ["./auto.sh"]
lance un script pour obtenir un fichier csv des temps de calculs du programme main.py (à modifier pour modification.py) avec plusieurs lancements, les n change à chaque lancement avec un incrément de 100 sur un intervalle de 100 à 3000 (utile pour retrouver les graphiques)

build du Dockerfile

docker build -t secret_generation_with_cryptographic_puzzles -f Dockerfile.python .

Lancement du build

docker run -it secret_generation_with_cryptographic_puzzles

Introduction

Le domaine des communications sécurisées est important dans la cybersécurité, où l'échange confidentiel d'informations reste un défi majeur. Dans ce contexte, le protocole de génération de secret par puzzles cryptographiques offre une approche pour assurer la confidentialité des communications. Ce protocole permet à deux entités, Alice et Bob, qui ne partagent initialement aucune information secrète, de générer un secret commun en communiquant sur un canal public. L'intérêt principal réside dans sa capacité à établir une

clé secrète partagée malgré la présence potentielle d'écouteurs, ou espions passifs, sans compromettre la sécurité des communications.

Le protocole exploite le concept de puzzles cryptographiques, où Alice génère une série de défis cryptographiques que Bob doit résoudre pour découvrir une clé cryptographique secrète. Dans notre implémentation, le protocole est conçu pour être robuste face à un adversaire passif, capable d'écouter, mais pas d'altérer ou d'injecter des messages dans la communication.

I. Description du Protocole

Principe Général

Le protocole de génération de secret par puzzles cryptographiques est conçu pour permettre à deux entités, Alice et Bob, de générer un secret commun en communiquant via un canal public potentiellement surveillé par un espion passif. L'idée fondamentale est d'utiliser des puzzles cryptographiques pour sécuriser l'échange de la clé secrète, sans que les parties aient besoin de partager des informations confidentielles à l'avance.

Le processus se déroule en plusieurs étapes :

Génération des Puzzles : Alice crée une série de puzzles cryptographiques. Chaque puzzle cache une clé secrète et un index. La clé de chaque puzzle est obtenue en hachant récursivement une clé préliminaire.

Envoi des Puzzles à Bob : Alice envoie la liste mélangée de ces puzzles à Bob par le biais d'un canal public.

Résolution du Puzzle par Bob : Bob choisit un puzzle au hasard, le résout pour découvrir la clé secrète et l'index associé, puis communique l'index à Alice.

Établissement du Secret Commun : Grâce à l'index, Alice identifie la clé secrète correspondante. Alice et Bob disposent maintenant d'une clé secrète commune pour sécuriser leurs communications ultérieures.

Rôles d'Alice et Bob

Alice (Générateur de Puzzles) : Son rôle est de générer les puzzles cryptographiques. Chaque puzzle est unique et contient une clé secrète destinée à être partagée avec Bob. Alice doit s'assurer que les puzzles sont suffisamment sécurisés pour résister à l'analyse d'un espion.

Bob (Solveur de Puzzles) : Bob choisit et résout un des puzzles envoyés par Alice. Le défi pour Bob est de déchiffrer correctement le puzzle pour obtenir la clé secrète et l'index, sans savoir à l'avance lequel des puzzles il a sélectionné.

Implémentation du Protocole

Description de la classe UserPuzzle :

La classe UserPuzzle est une implémentation programmée du protocole. Cette classe encapsule les fonctionnalités nécessaires pour qu'un utilisateur (Alice ou Bob) puisse générer ou résoudre les puzzles cryptographiques.

Fonctionnalités clés :

Génération de Puzzles : La classe permet de générer n puzzles, où chaque puzzle est associé à une clé secrète unique.

Elle utilise un algorithme de hachage cryptographique pour créer des clés de puzzle à partir de clés préliminaires.

Chiffrement des Puzzles : Chaque puzzle est chiffré en utilisant un algorithme de chiffrement symétrique (comme AES).

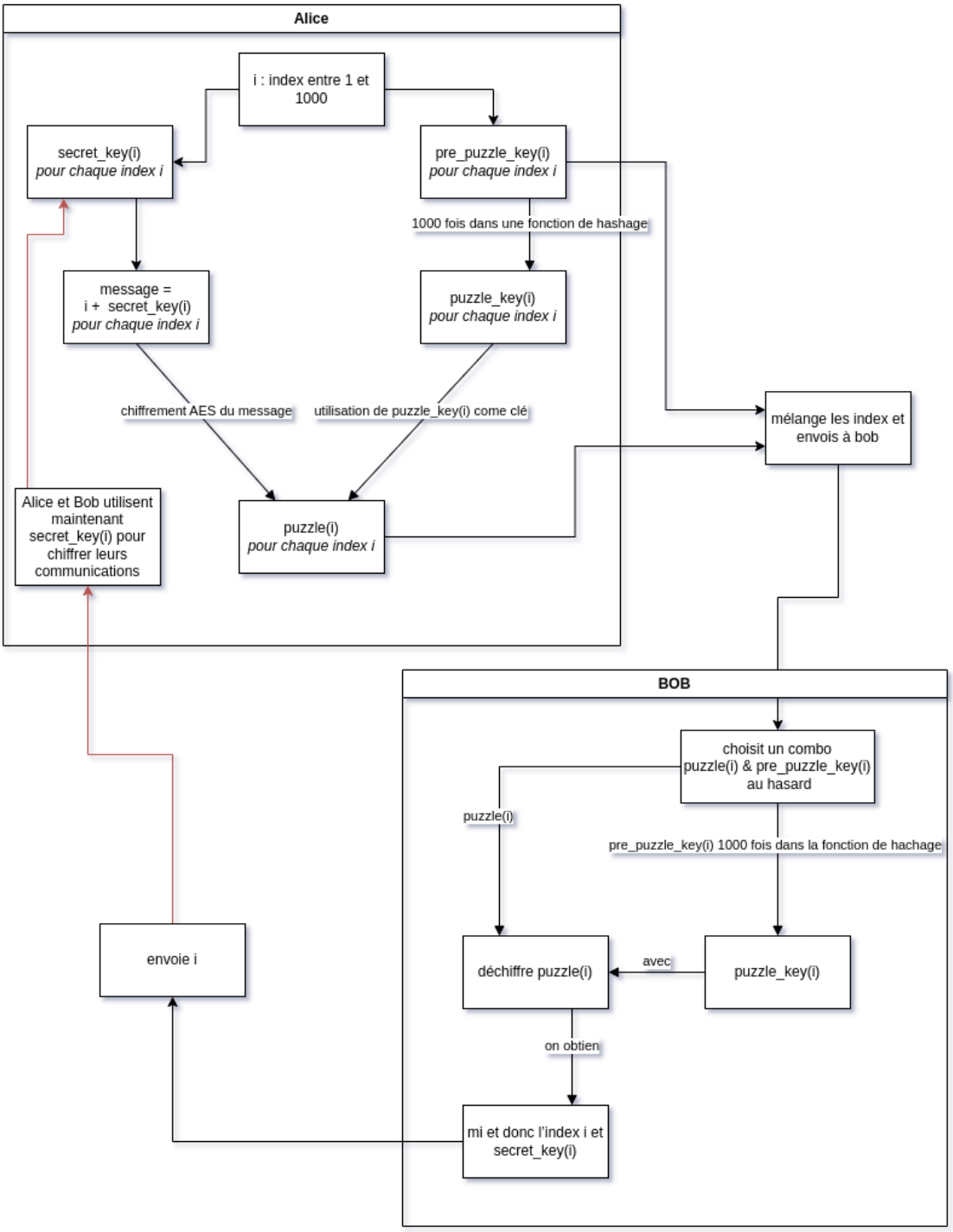
Le message chiffré comprend l'index du puzzle et la clé secrète associée.

Résolution de Puzzles : La classe offre la fonctionnalité pour choisir et résoudre un puzzle au hasard.

Elle déchiffre le puzzle sélectionné pour récupérer la clé secrète et l'index.

Gestion des Clés Secrètes : La classe maintient une correspondance entre les indices de puzzle et les clés secrètes.

Ceci permet à Alice de récupérer la clé secrète correspondant à l'index reçu de Bob.



En résumé, la classe UserPuzzle fournit une base solide pour simuler le protocole de génération de secret, en s'assurant que les principes de cryptographie nécessaires sont respectés et que les interactions entre Alice et Bob suivent le scénario prévu.

II. Analyse de la Charge de Travail

Charge de Travail pour Alice : Dans le protocole de génération de secret par puzzles cryptographiques, Alice, l'entité initiatrice, a pour rôle de générer un ensemble de n puzzles. Chaque puzzle est créé en suivant un processus en deux étapes : la génération d'une paire (pre_puzzle_key, secret_key) et le chiffrement de cette paire en utilisant un processus de hachage récursif et un algorithme de chiffrement symétrique.

Complexité en Termes de Calcul : Pour chaque puzzle, Alice effectue 1000 opérations de hachage, suivies d'une opération de chiffrement AES. La complexité de calcul pour chaque puzzle est donc dominée par les opérations de hachage. Étant donné que le hachage est répété 1000 fois pour chaque puzzle, la complexité totale en termes de calcul pour Alice est de l'ordre de O(1000n). Cependant, en pratique, les opérations de hachage sont relativement rapides et cette complexité reste gérable même pour de grandes valeurs de n.

Complexité en Termes d'Espace : Alice doit stocker les n paires (pre_puzzle_key, puzzle). Chaque clé et puzzle a une taille fixe (128 bits pour les clés et une taille déterminée par le chiffrement pour les puzzles). Ainsi, l'espace requis croît linéairement avec n, résultant en une complexité spatiale de O(n).

Charge de Travail pour Bob : Bob, de son côté, choisit un puzzle au hasard parmi les n disponibles et tente de le résoudre.

Résolution des Puzzles : Pour résoudre un puzzle, Bob doit d'abord calculer la puzzle_key en appliquant 1000 fois une fonction de hachage à pre_puzzle_key, puis déchiffrer le puzzle sélectionné. La complexité de cette tâche est donc O(1000) pour le hachage, plus la complexité du déchiffrement AES, qui est généralement considérée comme faible par rapport au hachage répété.

Comparaison avec Alice : Bien que Bob doive également effectuer 1000 opérations de hachage, il ne le fait que pour un seul puzzle. Ainsi, même si la charge de travail par puzzle est comparable entre Alice et Bob, la charge totale de travail pour Bob est nettement inférieure puisqu'elle est indépendante du nombre total de puzzles n.

Charge de Travail pour un Espion Potentiel : L'espion, passif, écoute les communications et tente de découvrir la clé secrète partagée entre Alice et Bob.

Difficulté à Briser la Sécurité du Protocole : L'espion doit résoudre au moins un des puzzles pour obtenir une clé secrète. Cela implique de tester chaque pre_puzzle_key avec le processus de hachage répété 1000 fois, suivi d'une tentative de déchiffrement, jusqu'à ce qu'un puzzle soit résolu. La complexité de cette tâche est donc de O(1000n) en termes de calcul, car l'espion doit potentiellement répéter ce processus pour chaque puzzle.

Méthodes Potentielles et Faisabilité : En théorie, l'espion peut essayer de résoudre tous les puzzles un par un jusqu'à trouver la bonne clé secrète. Cependant, cette approche est fortement dissuasive en pratique, surtout si le nombre de puzzles n est grand. Cela rend le protocole relativement sûr contre un espion passif, tant que le nombre de puzzles est

suffisamment élevé pour rendre la tâche de l'espion impraticable en termes de temps et de ressources de calcul.

Conclusion : la charge de travail pour Alice est significative, mais gérable, surtout compte tenu de son rôle d'initiatrice du protocole. Pour Bob, la charge de travail est beaucoup plus légère, car il ne doit résoudre qu'un seul puzzle. Enfin, pour un espion, bien que la méthode de briser la sécurité du protocole soit conceptuellement simple, elle est pratiquement irréalisable pour de grandes valeurs de n, offrant ainsi une protection efficace contre les écoutes passives.

III. Analyse de la Version Modifiée du Protocole

1. Modifications Apportées

Dans la version modifiée du protocole de génération de secret par puzzles cryptographiques, une modification a été apportée à l'étape 3. Initialement, Alice générait un ensemble de puzzles cryptographiques, chacun étant un couple de pre_puzzle_key et de puzzle chiffré, et envoyait cette liste mélangée à Bob. Dans la version modifiée, au lieu d'envoyer une liste unique de couples, Alice génère et mélange séparément deux listes : une contenant uniquement les pre_puzzle_key(i) et l'autre contenant les puzzle(i). Ces listes sont mélangées de manière indépendante et envoyées séparément à Bob.

2. Impact sur la Sécurité

Analyse de la Charge de Travail Supplémentaire pour Bob et un Espion

Pour Bob : Dans le protocole original, Bob choisit un puzzle au hasard et ne doit résoudre que ce puzzle spécifique. Dans la version modifiée, Bob doit d'abord calculer les puzzle_key pour tous les puzzles en appliquant le hachage récursif 1000 fois à chaque pre_puzzle_key. Ensuite, il doit essayer de déchiffrer un puzzle choisi en utilisant chacune de ces clés jusqu'à trouver la bonne. Cela augmente considérablement la charge de travail pour Bob, car il doit effectuer un nombre beaucoup plus important de déchiffrements avant de trouver la clé correcte.

Pour un Espion : Dans le protocole initial, un espion qui écoute la communication doit choisir un puzzle et le résoudre, similaire à Bob. Cependant, dans la version modifiée, l'espion est confronté à une tâche plus ardue. Il doit non seulement calculer tous les puzzle_key possibles mais aussi tenter de déchiffrer chaque puzzle avec chaque clé, ce qui représente une charge de travail considérablement accrue. Le nombre total d'opérations de déchiffrement que l'espion doit effectuer est exponentiellement plus élevé que dans le protocole original.

La version modifiée améliore la sécurité

La sécurité d'un système cryptographique est souvent mesurée par la quantité de travail nécessaire pour le briser. Dans la version modifiée du protocole, bien que la charge de

travail pour Bob augmente, la charge pour un espion augmente de manière beaucoup plus significative. Cette disproportion dans l'augmentation de la charge de travail rend le protocole plus résistant contre les attaques passives. En particulier, la difficulté pour un espion d'associer le bon pre_puzzle_key à son puzzle correspondant est considérablement accrue, ce qui améliore la sécurité globale du protocole contre l'écoute clandestine.

IV. Simulation et Résultats

Méthodologie de Simulation

Dans le cadre de notre analyse, nous avons effectué des simulations pour évaluer la performance et la sécurité du protocole de génération de secret par puzzles cryptographiques. Les simulations ont été structurées de manière à refléter les opérations d'Alice, de Bob et d'un espion potentiel, en se concentrant sur deux versions du protocole : la version originale et la version modifiée.

Alice : Pour chaque valeur de n (100, 1000, 10 000), Alice génère un ensemble correspondant de puzzles cryptographiques. Le temps requis pour cette génération est enregistré, en mettant l'accent sur le processus de hachage répété et le chiffrement AES.

Bob : Bob choisit et résout un puzzle au hasard dans la liste fournie par Alice. Le temps nécessaire pour sélectionner un puzzle, le hachage répété pour obtenir la clé de déchiffrement, et le déchiffrement du puzzle lui-même sont mesurés.

Résultats et Comparaison

Les résultats obtenus à partir de ces simulations sont les suivants :

Temps de Calcul pour Alice, Bob :

Pour n = 100, 1000, 10 000, le temps de calcul augmente de façon linéaire pour Alice et de manière exponentielle pour l'espion, surtout dans la version modifiée du protocole.

Bob présente un temps de calcul relativement constant dans la version originale, mais une augmentation notable dans la version modifiée.

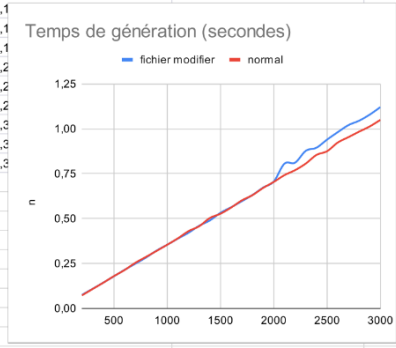
Comparaison avec Prédictions Théoriques :

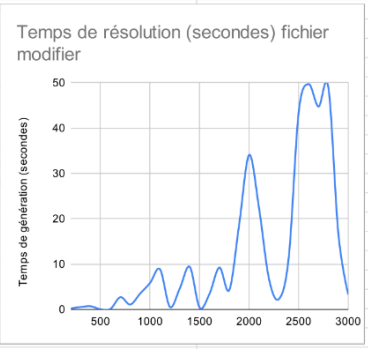
Les résultats observés pour Alice et Bob correspondent aux prédictions théoriques de complexité O(n).

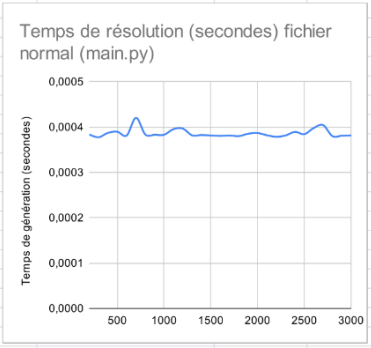
Pour l'espion, le temps de calcul s'accorde avec la complexité prévue, soulignant une difficulté nettement accrue pour la version modifiée du protocole.

Graphiques :

Temps de génération (secondes)	Temps de résolution (secondes)	main Temps de génération (secondes)	main Temps de résolution (secondes)	n
0,036442842	0,006287600001	0,036975829	0,0003981729988	100
0,075382813	0,230172252	0,072330801	0,0003836600008	200
0,805891856	22,29122747	0,742212977	0,0003819670001	2100
0,81179416	6,731606539	0,769890831	0,0003786030011	2200
0,877386733	2,272717587	0,806533415	0,0003815430009	2300
0,894764058	11,58881037	0,854410333	0,0003894110014	2400
0,940148139	43,52192765	0,875400853	0,0003844170005	2500
0,981505708	49,69861061	0,923661108	0,0003977979995	2600
1,020544553	44,7451772	0,953652209	0,0004045370006	2700
1,04507472	48,88113054	0,984333115	0,0003801130006	2800
1,079178028	16,76947659	1,013301287	0,0003808679994	2900
1,121174502	3,297512429	1,050035604	0,0003817429988	3000







Des graphiques sont utilisés pour illustrer visuellement la relation entre le temps de calcul et les différentes valeurs de n. Ces graphiques démontrent l'efficacité du protocole

Paramètres de Sécurité Raisonnables

À travers nos simulations, il apparaît que l'équilibre entre la sécurité et l'efficacité du protocole est atteint lorsque la valeur de n est suffisamment élevée pour rendre l'attaque coûteuse pour un espion, mais pas trop élevée pour surcharger Alice et Bob. Un n trop élevé peut entraîner des délais dans la génération et la résolution des puzzles, rendant le protocole impraticable en termes de performance.

Pour la version originale, un n autour de 1000 semble offrir un bon équilibre. Pour la version modifiée, un n légèrement plus élevé peut être envisagé, compte tenu de la sécurité accrue qu'elle offre (un n de 2000 pourrait être bien ?).

Ces paramètres doivent être ajustés en fonction des capacités de calcul disponibles pour les utilisateurs et les adversaires potentiels.

V. Importance de l'Espion Passif dans la Sécurité du Protocole

Rôle de l'Espion Passif

Le protocole de génération de secret par puzzles cryptographiques repose sur l'hypothèse d'un espion passif. Un espion passif est limité à l'écoute des communications entre Alice et

Bob sans altérer ni injecter de données. Cette hypothèse est cruciale pour la sécurité du protocole pour plusieurs raisons.

Impact d'un Espion Actif vs Passif

Espion Passif :

- Se limite à observer les échanges.
- Ne peut pas modifier les puzzles envoyés ni les réponses.
- La sécurité repose sur la difficulté de résoudre tous les puzzles, un défi faisable mais consommateur de temps pour l'espion.

Espion Actif :

- Peut intervenir dans la communication.
- Capable d'envoyer de faux puzzles ou de modifier les messages échangés.
- Peut induire Bob en erreur, compromettant la fiabilité du secret généré.

La sécurité n'est plus assurée, car l'espion actif peut briser le protocole non plus seulement par calcul, mais aussi par manipulation.

Vulnérabilité du Protocole face à des Attaques Actives

Le protocole est principalement vulnérable aux attaques actives où un espion peut :

Injecter des Puzzles Falsifiés : En remplaçant les puzzles originaux par les siens, l'espion peut contrôler la clé secrète partagée entre Alice et Bob.

Modifier les Messages : En altérant les communications, l'espion peut perturber la synchronisation du protocole, empêchant Alice et Bob de se mettre d'accord sur une clé secrète commune.

En conclusion, le protocole est conçu en tenant compte uniquement des menaces posées par un espion passif, assumant ainsi un environnement relativement contrôlé. Dans un contexte où des attaques actives sont possibles, le protocole nécessiterait des ajustements pour maintenir sa sécurité.

Conclusion

Résumé des Points Clés

Le protocole de génération de secret par puzzles cryptographiques offre une méthode pour deux parties, Alice et Bob, de générer un secret partagé malgré un canal de communication potentiellement compromis.

L'implémentation standard requiert un travail calculatoire linéaire de la part d'Alice et une charge raisonnable pour Bob, tandis que l'espion passif se heurte à une tâche significativement plus ardue pour compromettre la sécurité du protocole.

La version modifiée du protocole augmente la sécurité en complexifiant davantage la tâche de l'espion, bien qu'au détriment d'une charge de calcul accrue pour Bob.

Conclusions sur la Sécurité et l'Efficacité

Le protocole démontre une robustesse adéquate face à un espion passif, équilibrant la sécurité et l'efficacité pour Alice et Bob.

Toutefois, la version modifiée, bien que plus sécurisée, soulève des questions quant à l'efficacité en termes de temps de calcul pour Bob, en particulier pour des valeurs élevées de n.

Perspectives d'Amélioration ou d'Application

Une optimisation possible concerne l'équilibrage entre la charge de travail de Bob et la sécurité contre les attaques d'espionnage, potentiellement en ajustant le nombre de hachages.

L'application de ce protocole dans des contextes réels, comme les communications sécurisées dans les réseaux IoT ou les systèmes de messagerie, pourrait être envisagée, en tenant compte de leur contexte spécifique et des contraintes de performance.

En conclusion, bien que le protocole présente une approche solide pour la génération de secrets partagés, une attention particulière doit être portée à son équilibre entre sécurité et efficacité, surtout dans des scénarios d'application réelle où les ressources de calcul et les risques de sécurité varient grandement.

Annexe

main.py

```
import random
import hashlib
import os
import sys
import timeit
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad

class UserPuzzle:
   def __init__(self, n):
```

```
        print("Initialise l'utilisateur avec un nombre
de puzzles à générer.")

        self.n = n
        self.puzzles = []
        self.secret_keys = {}
        self.generate_puzzles()

    def hash_key(self, key):
        """
        Hash une clé 1000 fois en utilisant SHA-256.
        """
        for _ in range(1000):
            key = hashlib.sha256(key).digest()
        return key

    def generate_puzzles(self):

        print("Génère n puzzles cryptographiques.")
        print("Hash une clé 1000 fois en utilisant
SHA-256.")

        for i in range(1, self.n + 1):
            pre_puzzle_key = os.urandom(16)  # Génère
une chaîne aléatoire de 128 bits
            secret_key = os.urandom(16)      # Génère
une clé secrète aléatoire de 128 bits
            puzzle_key = self.hash_key(pre_puzzle_key)
            message = f"{i}:{secret_key.hex()}".encode()
            cipher = AES.new(puzzle_key, AES.MODE_CBC)
            iv = cipher.iv
            encrypted_message =
cipher.encrypt(pad(message, AES.block_size))
```

```
            self.puzzles.append((pre_puzzle_key, iv +
encrypted_message))
            self.secret_keys[i] = secret_key
        random.shuffle(self.puzzles)  # Mélange
aléatoirement la liste des puzzles


    def choose_puzzle(self):

        print("Choisi un puzzle au hasard.")

        return random.choice(self.puzzles)

    def solve_puzzle(self, pre_key, puzzle):

        print("Résout le puzzle pour obtenir l'indice et
la clé secrète.")

        puzzle_key = self.hash_key(pre_key)
        iv, encrypted_message = puzzle[:16], puzzle[16:]
        cipher = AES.new(puzzle_key, AES.MODE_CBC, iv)
        decrypted_message =
unpad(cipher.decrypt(encrypted_message),
AES.block_size).decode()
        index, secret_key_hex =
decrypted_message.split(':')
        return int(index), bytes.fromhex(secret_key_hex)

    def get_secret_key(self, index):

        print("Renvoie la clé secrète correspondant à
l'indice donné.")

        return self.secret_keys.get(index)
```

```
def main():
    # Nombre de puzzles à générer
    n = int(sys.argv[1])

    # Mesure du temps pour la génération des puzzles par
Alice
    start_time = timeit.default_timer()

    # Alice crée ses puzzles
    alice = UserPuzzle(n)

    # Mesure du temps pour la génération des puzzles par
Alice
    generation_time = timeit.default_timer() -
start_time

    # Mesure du temps pour la résolution des puzzles par
Bob
    start_time = timeit.default_timer()

    # Bob choisit et résout un puzzle au hasard
    pre_key, puzzle = alice.choose_puzzle()
    index, secret_key = alice.solve_puzzle(pre_key,
puzzle)

    # Mesure du temps pour la résolution des puzzles par
Bob
    resolution_time = timeit.default_timer() -
start_time

    # Vérification si Alice et Bob partagent le même
secret
    assert alice.get_secret_key(index) == secret_key
```

```
    print(f"Puzzle résolu! Index: {index}, Clé Secrète:
{secret_key.hex()}")

    print(f"Temps de génération des puzzles:
{generation_time} secondes")
    print(f"Temps de résolution des puzzles:
{resolution_time} secondes")
    #commande système pour ecrire dans un fichier
    os.system(f"echo {generation_time},
{resolution_time}, {n} >> resultats.csv")


if __name__ == "__main__":
    main()
```

modification.py

```
import random

import hashlib

import os

import sys

import timeit

from Crypto.Cipher import AES

from Crypto.Util.Padding import pad, unpad


class UtilisateurPuzzle:

    def __init__(self, n):
```

```
        self.n = n
        self.pre_cles_puzzle = []  # Liste des clés de pré-puzzle
        self.puzzles = []  # Liste des énigmes
        self.cles_secretes = {}  # Dictionnaire pour stocker les clés secrètes
        self.generer_puzzles()

    def hash_key(self, cle):
        for _ in range(1000):
            cle = hashlib.sha256(cle).digest()
        return cle

    def generer_puzzles(self):
        print("Initialisation de l'utilisateur avec un nombre de puzzles à générer.")
        for i in range(1, self.n + 1):
            pre_cle_puzzle = os.urandom(16)  # Chaîne aléatoire de 128 bits
            cle_secrete = os.urandom(16)  # Clé secrète aléatoire de 128 bits
            cle_puzzle = self.hash_key(pre_cle_puzzle)
            message = f"{i}:{cle_secrete.hex()}".encode()
```

```
            cipher = AES.new(cle_puzzle, AES.MODE_CBC)
            iv = cipher.iv
            message_chiffre =
cipher.encrypt(pad(message, AES.block_size))
            self.pre_cles_puzzle.append(pre_cle_puzzle)
            self.puzzles.append(iv + message_chiffre)
            self.cles_secretes[i] = cle_secrete

        random.shuffle(self.pre_cles_puzzle)
        random.shuffle(self.puzzles)

    def choisir_et_resoudre_puzzle(self):
        print("Choix d'un puzzle au hasard.")
        puzzle_choisi = random.choice(self.puzzles)
        for pre_cle in self.pre_cles_puzzle:
            cle_puzzle = self.hash_key(pre_cle)
            for i in range(self.n):
                try:
                    iv, message_chiffre =
puzzle_choisi[:16], puzzle_choisi[16:]
                    cipher = AES.new(cle_puzzle,
AES.MODE_CBC, iv)
```

```
                    message_dechiffre =
unpad(cipher.decrypt(message_chiffre),
AES.block_size).decode()

                    index, cle_secrete_hex =
message_dechiffre.split(':')

                    if self.cles_secretes[int(index)] ==
bytes.fromhex(cle_secrete_hex):

                        print(f"Puzzle résolu! Indice :
{index}, Clé Secrète : {cle_secrete_hex}")

                        return int(index),
bytes.fromhex(cle_secrete_hex)

                except ValueError:

                    continue  # Continuer à essayer avec
d'autres clés si le déchiffrement échoue

        raise ValueError("Impossible de résoudre aucun
puzzle")



    def obtenir_cle_secrete(self, index):

        """

        Retourne la clé secrète correspondant à l'indice
donné.

        """

        return self.cles_secretes.get(index)



def main():
```

```
    # premier paramètre quand on lance le script
    n = int(sys.argv[1])

    # Mesure du temps pour la génération des puzzles par Alice
    start_time = timeit.default_timer()

    # Alice crée ses énigmes
    alice = UtilisateurPuzzle(n)

    # Mesure du temps pour la génération des puzzles par Alice
    generation_time = timeit.default_timer() - start_time
    # Mesure du temps pour la résolution des puzzles par Bob
    start_time = timeit.default_timer()

    # Bob choisit et résout une énigme au hasard
    index, cle_secrete = alice.choisir_et_resoudre_puzzle()
```

```
    # Mesure du temps pour la résolution des puzzles par Bob
    resolution_time = timeit.default_timer() - start_time

    # Vérifier si Alice et Bob partagent la même clé secrète
    assert alice.obtenir_cle_secrete(index) == cle_secrete

    print(f"Temps de génération des puzzles: {generation_time} secondes n = {n}")
    print(f"Temps de résolution des puzzles: {resolution_time} secondes n = {n}")
    #commande système pour ecrire dans un fichier
    os.system(f"echo {generation_time}, {resolution_time}, {n} >> resultats.csv")


if __name__ == "__main__":
    main()
```