Compte rendu du mini-projet 1

Cryptographie et sécurité des données

Réalisé par : FADDAK Mehdi
OUCHEN Younes
Sous l'encadrement de: AZBEG Kebira

Année universitaire : 2025 - 2026

Table des matières

I - Contexte :

Dans le cadre de ce projet, il s'agit de concevoir et de développer un système complet de sécurisation des échanges de fichiers entre deux utilisateurs, en s'appuyant sur les principes fondamentaux de la cryptographie asymétrique et symétrique. L'objectif principal est de protéger les échanges contre toute interception, modification ou falsification, tout en assurant la fiabilité de l'expéditeur. Ainsi, le système devra garantir l'authenticité, l'intégrité, la confidentialité et la non-répudiation des données échangées. Concrètement, cela implique que chaque fichier transmis soit signé par l'expéditeur pour prouver son identité, chiffré pour empêcher toute lecture non autorisée, et accompagné des éléments nécessaires à sa vérification et à son déchiffrement par le destinataire.

II - Contraintes et objectifs

Pour atteindre ces objectifs, plusieurs contraintes techniques sont imposées. Le chiffrement asymétrique reposera sur l'algorithme RSA-2048, utilisé à la fois pour la signature numérique et pour le chiffrement de la clé symétrique. Le contenu des fichiers sera protégé grâce à AES-128 en mode CBC, garantissant la confidentialité des données. L'intégrité sera assurée par le calcul d'empreintes numériques à l'aide de SHA-256, tandis que l'ensemble du processus sera implémenté en Python, en s'appuyant sur la bibliothèque PyCryptodome. Le système devra donc permettre la génération, la signature, le chiffrement et la vérification sécurisée de fichiers dans un flux complet d'échange entre deux utilisateurs.

III - Technique et technologie

1. RSA-2048 (cryptographie asymétrique)

RSA est un algorithme de cryptographie asymétrique reposant sur un couple de clés : une clé publique (pour chiffrer ou vérifier une signature) et une clé privée (pour déchiffrer ou signer). Sa sécurité repose sur la difficulté de factoriser de grands nombres premiers. L'utilisation de RSA-2048 signifie que la taille de la clé est de 2048 bits, offrant un très bon niveau de sécurité pour les échanges modernes. Dans ce projet, RSA servira à deux fins :

Signer le fichier avec la clé privée de l'expéditeur pour assurer l'authenticité et la non-répudiation.

Chiffrer la clé symétrique AES avec la clé publique du destinataire, garantissant que seul ce dernier puisse la déchiffrer.

2. AES-128 en mode CBC (cryptographie symétrique)

AES (Advanced Encryption Standard) est un algorithme de cryptographie symétrique, ce qui signifie qu'il utilise la même clé pour le chiffrement et le déchiffrement. L'AES-128 utilise une clé de 128 bits, assurant un excellent équilibre entre performance et sécurité. Le mode CBC (Cipher Block Chaining) ajoute un niveau de protection supplémentaire en rendant chaque bloc de texte chiffré dépendant du précédent, grâce à un vecteur d'initialisation (IV). Cela empêche la répétition de motifs dans le texte chiffré et renforce la confidentialité des données.

3. SHA-256 (empreinte numérique / hachage)

SHA-256 est une fonction de hachage cryptographique appartenant à la famille SHA-2. Elle transforme n'importe quel message en une empreinte (ou "hash") unique de 256 bits. Cette empreinte est irréversible et unique à chaque contenu : le moindre changement dans le fichier modifie totalement le hash. Dans ce projet, SHA-256 est utilisé pour vérifier l'intégrité du fichier et pour générer une empreinte avant la signature RSA, garantissant que le fichier n'a pas été altéré.

4. PyCryptodome (bibliothèque Python de cryptographie)

pycryptodome

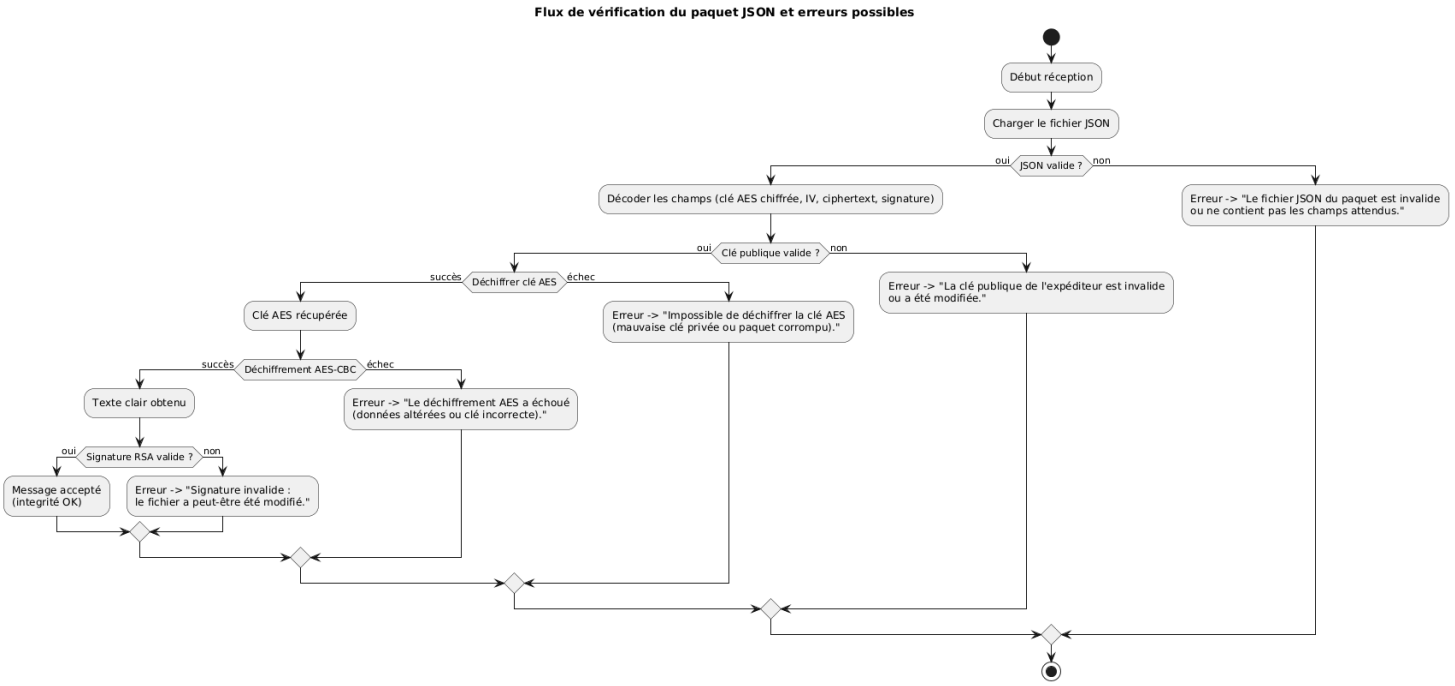
PyCryptodome est une bibliothèque Python moderne qui fournit des outils complets pour implémenter la cryptographie symétrique, asymétrique, le hachage et la signature numérique. Elle est conçue comme une alternative améliorée à PyCrypto. Dans ce projet, PyCryptodome permettra d'utiliser de manière simple et sécurisée les algorithmes RSA, AES et SHA-256, tout en gérant la génération des clés, le chiffrement/déchiffrement des données et la création/vérification des signatures numériques.

5. Python (langage de programmation)

Python est un langage de programmation polyvalent, clair et lisible, très utilisé dans le domaine de la sécurité informatique et de la cryptographie. Sa syntaxe simple et sa large communauté facilitent la mise en œuvre d'algorithmes complexes comme RSA, AES ou SHA-256. De plus, son écosystème riche en bibliothèques (comme PyCryptodome) en fait un choix idéal pour développer rapidement des applications de sécurité, de chiffrement de données et d'échanges sécurisés de fichiers, tout en garantissant une bonne maintenabilité du code.

IV - Fonctionnement

1. Diagramme d’activité



2. Explication

Le diagramme ci-dessus illustre en détail le processus de vérification, de déchiffrement et de validation d’un fichier sécurisé transmis sous forme de paquet JSON. Ce flux représente l’ensemble des étapes effectuées par le destinataire pour garantir la validité, l’intégrité et l’authenticité des données reçues.

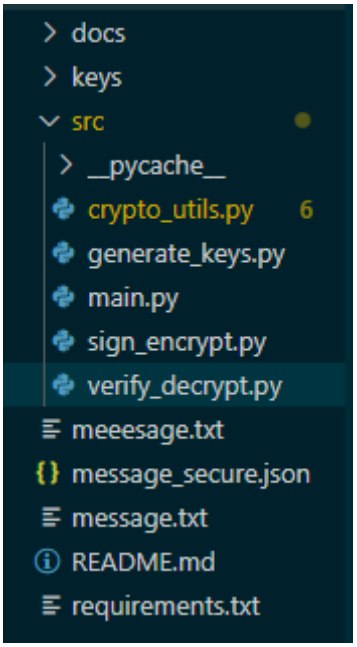
Le processus débute par la réception du fichier JSON, suivie d’une vérification de sa validité syntaxique et structurelle. Si le fichier ne contient pas les champs attendus (clé AES chiffrée, IV, texte chiffré et signature), une erreur est immédiatement générée signalant que le JSON est invalide. Si le fichier est valide, les différents champs sont extraits et le système tente ensuite de déchiffrer la clé AES à l’aide de la clé privée RSA du destinataire. En cas d’échec (clé privée incorrecte ou paquet corrompu), le processus s’interrompt avec un message d’erreur approprié.

Une fois la clé AES récupérée avec succès, le module procède au déchiffrement du contenu chiffré (ciphertext) en utilisant l’algorithme AES en mode CBC et le vecteur d’initialisation (IV). Si cette étape échoue — par exemple à cause d’une clé incorrecte ou de données altérées —, une erreur signale l’échec du déchiffrement. Si le déchiffrement est réussi, on obtient alors le texte clair du message original.

La dernière étape consiste à vérifier la signature RSA de l'expéditeur, en la comparant à l'empreinte SHA-256 calculée à partir du texte clair. Si la signature est valide, le message est accepté : cela confirme que son intégrité est préservée et qu'il provient bien de l'expéditeur authentique. Dans le cas contraire, une erreur indique une signature invalide, laissant supposer que le fichier a pu être modifié ou falsifié pendant la transmission.

V - Application

Structure



Le projet est composé de plusieurs scripts python qu'on détaillera par la suite.

1. Etape - 1: Génération des clés RSA

Ce processus est assuré par le fichier **generate_keys.py:**

```python
def parse_args() -> argparse.Namespace:
    parser = argparse.ArgumentParser(
        description="Génère une paire de clés RSA-2048 pour signer et chiffrer.",
    )
    parser.add_argument(
        "--output-dir",
        type=Path,
        default=Path("keys"),
        help="Dossier où enregistrer les clés (créé automatiquement si besoin).",
    )
    parser.add_argument(
        "--prefix",
        default="id_rsa",
        help="Préfixe des fichiers générés (ex: id_rsa -> id_rsa_private.pem).",
    )
    parser.add_argument(
        "--passphrase",
        default=None,
        help="Mot de passe optionnel pour chiffrer la clé privée (PKCS#8 + scrypt+AES128).",
    )
    return parser.parse_args()
```

```python
def main() -> None:
    args = parse_args()
    output_dir: Path = args.output_dir
    output_dir.mkdir(parents=True, exist_ok=True)

    private_key = generate_rsa_keypair()
    private_path = output_dir / f"{args.prefix}_private.pem"
    public_path = output_dir / f"{args.prefix}_public.pem"

    passphrase: Optional[str] = args.passphrase
    export_private_key(private_key, private_path, passphrase=passphrase)
    export_public_key(private_key.public_key(), public_path)

    print(" Paire de clés générée !")
    print(f"  . Clé privée : {private_path}")
    print(f"  . Clé publique : {public_path}")
    if passphrase:
        print("  Garde bien le mot de passe, il sera demandé pour utiliser la clé privée.")


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

Ce script génère une paire de clés RSA-2048, sauvegarde la clé privée et la clé publique dans des fichiers PEM, et permet de protéger la clé privée avec un mot de passe optionnel.

Le code est un utilitaire pour la génération de clés RSA destinées à signer et chiffrer des fichiers dans un projet de sécurité. Il utilise **argparse** pour récupérer les options de l'utilisateur, telles que le répertoire de sortie des clés, le préfixe des fichiers et une passphrase pour chiffrer la clé privée. La fonction principale crée le dossier de sortie si nécessaire, génère une clé privée RSA-2048 via **generate_rsa_keypair()**, et exporte la clé privée (éventuellement protégée par la passphrase) et la clé publique associée dans des fichiers PEM. Enfin, le script affiche à l'écran le chemin des fichiers générés et rappelle à l'utilisateur de conserver la passphrase si elle a été utilisée, assurant ainsi la sécurité et la confidentialité de la clé privée.

2. Etape - 2: Signatures et chiffrement (Binôme A)

Ce processus est assuré par le fichier **sign_encrypt.py:**

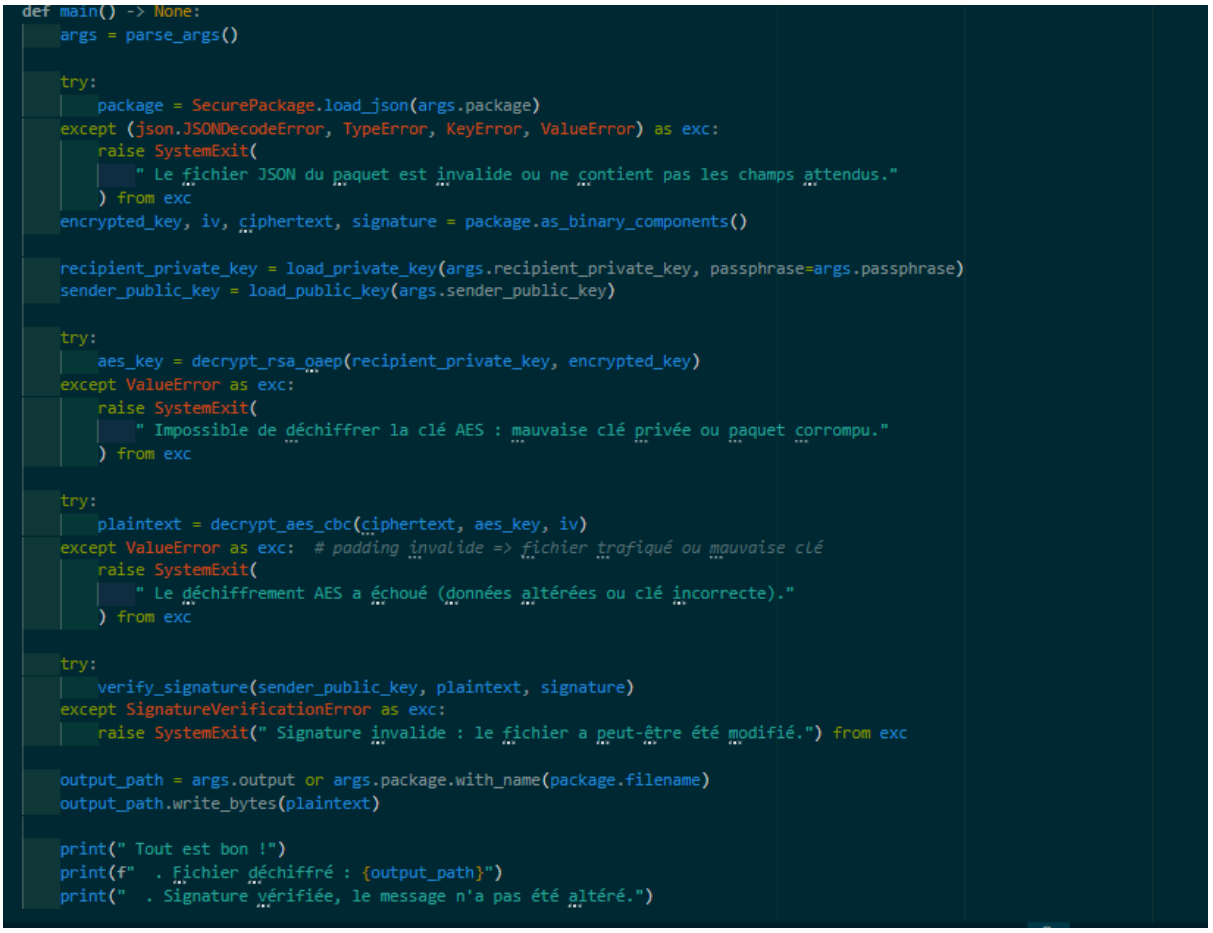
```python
def parse_args() -> argparse.Namespace: ···


def main() -> None:
    args = parse_args()

    plaintext = args.input.read_bytes()
    sender_private_key = load_private_key(args.sender_private_key, passphrase=args.passphrase)
    recipient_public_key = load_public_key(args.recipient_public_key)

    signature = sign_bytes(sender_private_key, plaintext)
    ciphertext, aes_key, iv = encrypt_aes_cbc(plaintext)
    encrypted_key = encrypt_rsa_oaep(recipient_public_key, aes_key)

    package = SecurePackage.from_components(
        encrypted_key=encrypted_key,
        iv=iv,
        ciphertext=ciphertext,
        signature=signature,
        filename=args.input.name,
    )
    package.to_json(args.output)

    print(" Paquet sécurisé prêt !")
    print(f"  . Fichier JSON : {args.output}")
    print("  . Contient la signature, la clé AES chiffrée, l'IV et le ciphertext.")
    print("Envoie ce JSON au destinataire ainsi que ta clé publique.")


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

Ce script signe un fichier avec la clé privée de l'expéditeur, chiffre son contenu en utilisant AES-128, protège la clé AES avec la clé publique du destinataire et génère un paquet sécurisé au format JSON pour un échange confidentiel et authentifié.

Le code implémente un chiffrement hybride pour sécuriser un fichier avant transmission. Il commence par lire le fichier en clair et charger la clé privée de l'expéditeur ainsi que la clé publique du destinataire. Le fichier est d'abord signé numériquement avec RSA-PKCS#1 v1.5 pour garantir l'authenticité et l'intégrité. Ensuite, le contenu est chiffré symétriquement avec AES-128 en mode CBC pour assurer la confidentialité, et la clé AES générée est chiffrée asymétriquement avec RSA OAEP afin que seul le destinataire puisse la récupérer. Tous ces éléments (ciphertext, clé AES chiffrée, IV, signature et métadonnées) sont encapsulés dans un objet **SecurePackage** et exportés en JSON, créant ainsi un paquet sécurisé prêt à être envoyé au destinataire, qui pourra ensuite le déchiffrer et vérifier la signature.

3. Etape - 3/4 : Déchiffrement et vérification (Binôme B) et validation

Ce processus est assuré par le fichier **verify_decrpyt.py :**



```python
def main() -> None:
    args = parse_args()

    try:
        package = SecurePackage.load_json(args.package)
    except (json.JSONDecodeError, TypeError, KeyError, ValueError) as exc:
        raise SystemExit(
            " Le fichier JSON du paquet est invalide ou ne contient pas les champs attendus."
        ) from exc
    encrypted_key, iv, ciphertext, signature = package.as_binary_components()

    recipient_private_key = load_private_key(args.recipient_private_key, passphrase=args.passphrase)
    sender_public_key = load_public_key(args.sender_public_key)

    try:
        aes_key = decrypt_rsa_oaep(recipient_private_key, encrypted_key)
    except ValueError as exc:
        raise SystemExit(
            " Impossible de déchiffrer la clé AES : mauvaise clé privée ou paquet corrompu."
        ) from exc

    try:
        plaintext = decrypt_aes_cbc(ciphertext, aes_key, iv)
    except ValueError as exc:  # padding invalide => fichier trafiqué ou mauvaise clé
        raise SystemExit(
            " Le déchiffrement AES a échoué (données altérées ou clé incorrecte)."
        ) from exc

    try:
        verify_signature(sender_public_key, plaintext, signature)
    except SignatureVerificationError as exc:
        raise SystemExit(" Signature invalide : le fichier a peut-être été modifié.") from exc

    output_path = args.output or args.package.with_name(package.filename)
    output_path.write_bytes(plaintext)

    print(" Tout est bon !")
    print(f"  . Fichier déchiffré : {output_path}")
    print("  . Signature vérifiée, le message n'a pas été altéré.")
```

Ce script déchiffre un paquet sécurisé JSON reçu, récupère la clé AES chiffrée, déchiffre le contenu avec AES-128, et vérifie la signature RSA de l'expéditeur pour assurer l'authenticité et l'intégrité du fichier.

Le code implémente le processus de réception et de traitement d'un fichier sécurisé transmis en mode hybride. Il commence par charger le paquet JSON contenant le ciphertext, la clé AES chiffrée, l'IV et la signature, puis convertit ces champs en données binaires. La clé AES est déchiffrée asymétriquement avec la clé privée du destinataire, puis le fichier est déchiffré symétriquement en utilisant AES-128-CBC. Ensuite, la signature numérique est vérifiée avec la clé publique de l'expéditeur pour confirmer que le fichier n'a pas été modifié et provient bien de l'émetteur attendu. Enfin, le fichier en clair est écrit sur le disque, et le script informe l'utilisateur que le déchiffrement et la vérification ont réussi, garantissant ainsi la confidentialité, l'intégrité et l'authenticité des données reçues.

4. Main : cohésion de toute les étapes précédentes

Ce processus est assuré par le fichier **main.py** :

```
def parse_args() -> argparse.Namespace:
    description="Demande un nom de fichier + contenu, puis signe et chiffre tout ça.",
    )
    parser.add_argument(
        "--sender-private-key",
        type=Path,
        required=True,
        help="Clé privée RSA de l'expéditeur (PEM).",
    )
    parser.add_argument(
        "--recipient-public-key",
        type=Path,
        required=True,
        help="Clé publique RSA du destinataire (PEM).",
    )
    parser.add_argument(
        "--package",
        type=Path,
        default=Path("message_secure.json"),
        help="Nom du paquet JSON produit (défaut : message_secure.json).",
    )
    parser.add_argument(
        "--passphrase",
        default=None,
        help="Mot de passe pour la clé privée si elle est chiffrée.",
    )
    parser.add_argument(
        "--save-plaintext",
        action="store_true",
        help="Sauvegarde aussi le fichier en clair localement.",
    )
    return parser.parse_args()
```

```python
def _derive_public_key_path(private_key_path: Path) -> Path:
    if private_key_path.suffix:
        base = private_key_path.stem
        suffix = private_key_path.suffix
    else:
        base = private_key_path.name
        suffix = ""
    if base.endswith("_private"):
        base = base[:-8] + "_public"
    else:
        base = base + "_public"
    return private_key_path.with_name(base + (suffix or ".pem"))


def _derive_private_key_path(public_key_path: Path) -> Path:
    if public_key_path.suffix:
        base = public_key_path.stem
        suffix = public_key_path.suffix
    else:
        base = public_key_path.name
        suffix = ""
    if base.endswith("_public"):
        base = base[:-7] + "_private"
    else:
        base = base + "_private"
    return public_key_path.with_name(base + (suffix or ".pem"))
```

```python
def ensure_sender_private_key(path: Path, passphrase: Optional[str]) -> Path:
    if path.exists():
        return path

    path.parent.mkdir(parents=True, exist_ok=True)
    key = generate_rsa_keypair()
    export_private_key(key, path, passphrase=passphrase)
    public_path = _derive_public_key_path(path)
    export_public_key(key.public_key(), public_path)
    print(" Aucune clé privée expéditeur trouvée, on en génère une nouvelle :")
    print(f"  • Clé privée : {path}")
    print(f"  • Clé publique : {public_path}")
    return path


def ensure_recipient_public_key(path: Path) -> Path:
    if path.exists():
        return path

    path.parent.mkdir(parents=True, exist_ok=True)
    key = generate_rsa_keypair()
    private_path = _derive_private_key_path(path)
    export_public_key(key.public_key(), path)
    export_private_key(key, private_path)
    print(" Pas de clé publique destinataire détectée, génération d'un nouveau couple :")
    print(f"  • Clé publique : {path}")
    print(f"  • Clé privée (à garder secrète) : {private_path}")
    print("  → Partage cette nouvelle clé publique avec l'expéditeur.")
    return path
```

```python
def capture_plaintext() -> tuple[str, bytes]:
    filename = ""
    while not filename:
        filename = input("Nom du fichier (ex: message.txt) : ").strip()
        if not filename:
            print("  Le nom ne peut pas être vide.")

    print("Écris ton texte ci-dessous. Tape juste 'EOF' sur une ligne pour terminer :")
    lines: list[str] = []
    while True:
        try:
            line = input()
        except EOFError:
            break
        if line == "EOF":
            break
        lines.append(line)
    plaintext = "\n".join(lines).encode("utf-8")
    return filename, plaintext
```

```python
def main() -> None:
    args = parse_args()

    sender_private_key_path = ensure_sender_private_key(args.sender_private_key, args.passphrase)
    recipient_public_key_path = ensure_recipient_public_key(args.recipient_public_key)

    filename, plaintext = capture_plaintext()

    sender_private_key = load_private_key(sender_private_key_path, passphrase=args.passphrase)
    recipient_public_key = load_public_key(recipient_public_key_path)

    signature = sign_bytes(sender_private_key, plaintext)
    ciphertext, aes_key, iv = encrypt_aes_cbc(plaintext)
    encrypted_key = encrypt_rsa_oaep(recipient_public_key, aes_key)

    package = SecurePackage.from_components(
        encrypted_key=encrypted_key,
        iv=iv,
        ciphertext=ciphertext,
        signature=signature,
        filename=filename,
    )
    package.to_json(args.package)

    if args.save_plaintext:
        Path(filename).write_bytes(plaintext)
        print(f"  Fichier en clair enregistré : {filename}")

    print(" Paquet sécurisé créé !")
    print(f"  • JSON : {args.package}")
    print("  • À transmettre au destinataire (avec ta clé publique).")
```

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

Le code implémente un utilitaire interactif pour créer des messages sécurisés. Il commence par vérifier que les clés RSA de l'expéditeur et du destinataire existent, en générant de nouvelles clés si nécessaire. L'utilisateur est ensuite invité à saisir un nom de fichier et le contenu du message, qui est capturé jusqu'à la ligne « EOF ». Le message est signé numériquement avec la clé privée de l'expéditeur pour garantir l'authenticité et l'intégrité, puis chiffré symétriquement avec AES-128-CBC. La clé AES est ensuite chiffrée asymétriquement avec la clé publique du destinataire pour que seul ce dernier puisse déchiffrer le message. Tous les éléments (ciphertext, clé AES chiffrée, IV, signature et métadonnées) sont encapsulés dans un objet SecurePackage et exportés en JSON. Le script peut également sauvegarder le fichier en clair localement si demandé, et affiche à l'utilisateur les chemins du fichier JSON généré et, le cas échéant, du fichier en clair. Ce processus assure ainsi la confidentialité, l'intégrité et l'authenticité du message avant transmission.

VI - Exécution

1. Chiffrer les données:

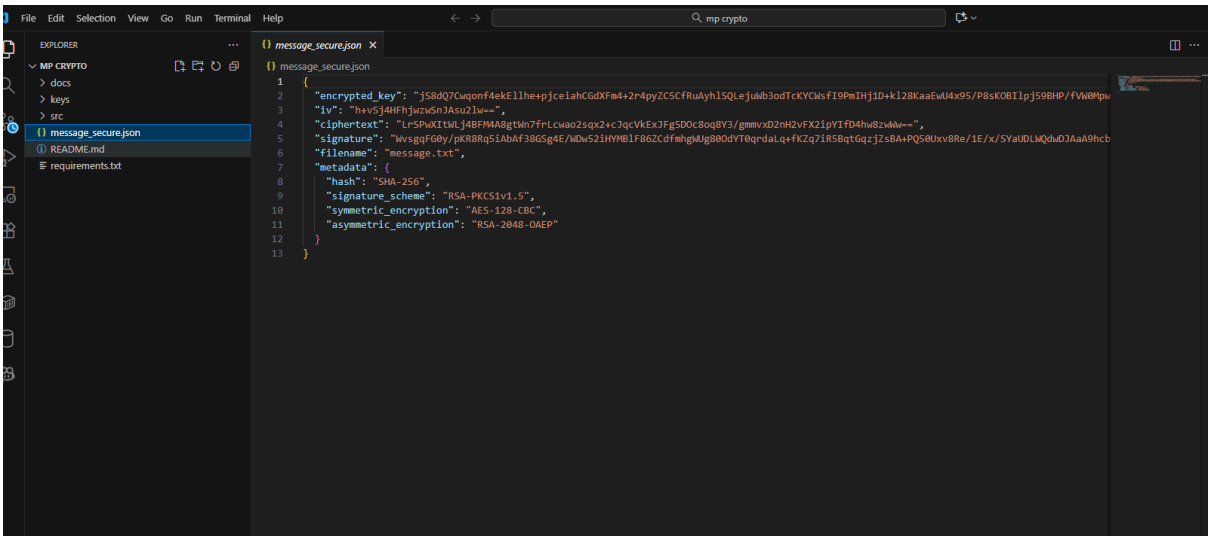
On execute **main.py** ou on va définir un fichier texte message.txt avec un message simple et EOF pour la fin de fichier. Cette commande peut être accompagnée d'une passphrase pour la clé privé binomeA, ou pour sauvegarder le fichier créé localement pour test:

- python src\main.py --sender-private-key keys\binomeA_private.pem --recipient-public-key keys\binomeB_public.pem --package message_secure.json

```
C:\Users\hp\Documents\Projects\mp crypto>python src\main.py --sender-private-key keys\binomeA_private.
pem --recipient-public-key keys\binomeB_public.pem --package message_secure.json
Nom du fichier (ex: message.txt) : message.txt
Écris ton texte ci-dessous. Tape juste 'EOF' sur une ligne pour terminer :
ce message doit etre chiffré et converti en json
EOF
 Paquet sécurisé créé !
  • JSON : message_secure.json
  • À transmettre au destinataire (avec ta clé publique).

C:\Users\hp\Documents\Projects\mp crypto>
C:\Users\hp\Documents\Projects\mp crypto>
```

NB: On peut aussi directement utiliser sign_encrypt.py avec le fichier comme paramètre et la clé privé A et publique B généré par generate_keys.py

Voici le contenu de **message_secure.json:**



2. Déchiffrement et vérification

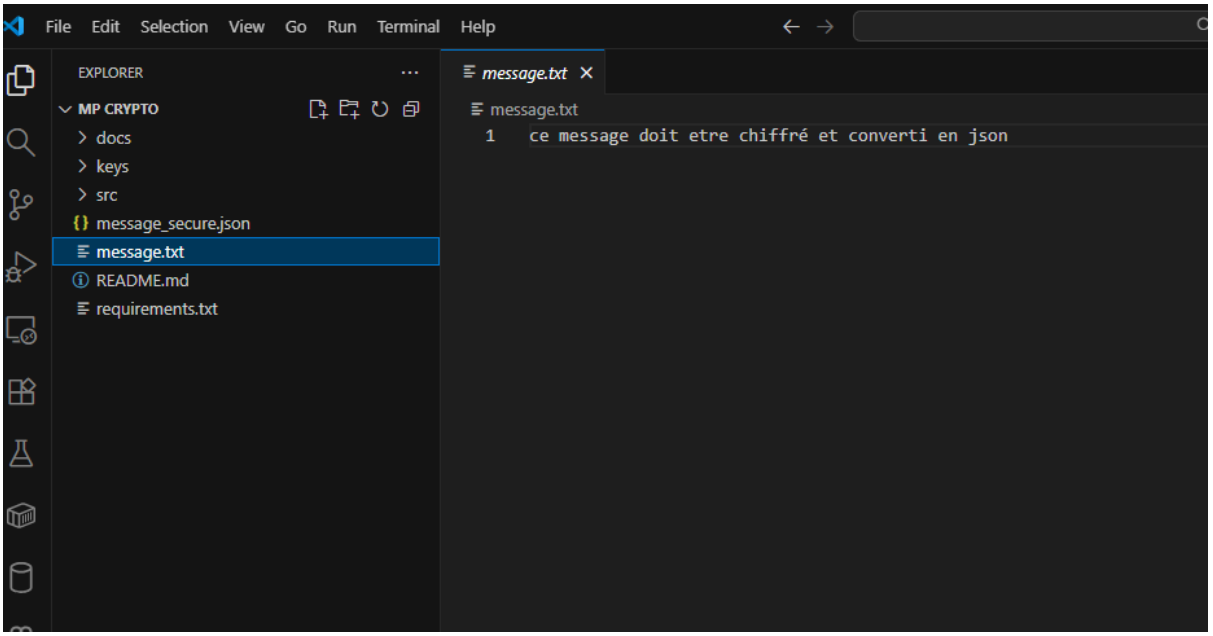
Pour déchiffrer et vérifier on exécute la commande suivante avec la clé privé B et clé publique A:

- verify_decrypt.py --package message_secure.json --recipient-private-key keys\binomeB_private.pem --sender-public-key keys\binomeA_public.pem

En cas de conservation du fichier json on reçoit:

```
C:\Users\hp\Documents\Projects\mp crypto>python src\verify_decrypt.py --package message_secure.json --
recipient-private-key keys\binomeB_private.pem --sender-public-key keys\binomeA_public.pem
 Tout est bon !
  . Fichier déchiffré : message.txt
  . Signature vérifiée, le message n'a pas été altéré.

C:\Users\hp\Documents\Projects\mp crypto>
```

Ainsi que la création du fichier.



En cas de modification de clé publique A

```
C:\Users\hp\Documents\Projects\mp crypto>python src\verify_decrypt.py --package message_secure.json --
recipient-private-key keys\binomeB_private.pem --sender-public-key keys\binomeA_public.pem
 La clé publique de l'expéditeur est invalide ou a été modifiée. Vérifiez que vous utilisez la bonne c
lé.

C:\Users\hp\Documents\Projects\mp crypto>
```

En cas de changement du contenu de json (Clé chiffré ou ciphertext ou signature)

```
C:\Users\hp\Documents\Projects\mp crypto>python src\verify_decrypt.py --package message_secure.json --
recipient-private-key keys\binomeB_private.pem --sender-public-key keys\binomeA_public.pem
 Signature invalide : le fichier a peut-être été modifié.

C:\Users\hp\Documents\Projects\mp crypto>python src\verify_decrypt.py --package message_secure.json --
recipient-private-key keys\binomeB_private.pem --sender-public-key keys\binomeA_public.pem
 Signature invalide : le fichier a peut-être été modifié.

C:\Users\hp\Documents\Projects\mp crypto>python src\verify_decrypt.py --package message_secure.json --
recipient-private-key keys\binomeB_private.pem --sender-public-key keys\binomeA_public.pem
 Impossible de déchiffrer la clé AES : mauvaise clé privée ou paquet corrompu.

C:\Users\hp\Documents\Projects\mp crypto>
```

VII - Conclusion

Le projet implémente un système complet d'échange de fichiers sécurisé basé sur une approche hybride combinant chiffrement symétrique (AES-128-CBC) et asymétrique (RSA-2048), accompagné de signatures numériques RSA pour garantir l'intégrité et l'authenticité des données. Les différents scripts permettent de couvrir toutes les étapes du processus : génération de clés RSA, signature et chiffrement des fichiers ou messages par l'expéditeur, transmission sous forme de paquets JSON sécurisés, et réception avec déchiffrement et vérification de signature par le destinataire. L'utilisation de classes comme SecurePackage et de formats JSON rend le système portable et lisible, tandis que la gestion des clés (passphrase, export/import) renforce la sécurité opérationnelle. Finalement, le projet illustre clairement les principes fondamentaux de la cryptographie appliquée à la protection des échanges numériques, en offrant une solution fonctionnelle, sécurisée et facilement extensible pour des communications confidentielles entre utilisateurs.