Un journaliste souhaite mettre à disposition de sa « source » un outil de chiffrement simple d'utilisation permettant de sécuriser les informations qu'elle lui envoie. L'application `POC` aura pour fonction de chiffrer le fichier qui lui est indiqué en paramètre selon un système hybride classique (fig. 40) :

— tout d'abord une *clef de session AES de 16 octets* sera choisie aléatoirement ;
— le fichier indiqué sera chiffré en AES avec la clef choisie en mode CBC et avec bourrage PKCS5 en s'appuyant sur un *vecteur d'initialisation de 16 octets*, lui aussi choisi aléatoirement ;
— la clef AES utilisée sera chiffrée en RSA avec bourrage PKCS1 en utilisant une clef publique fixée dont *le module public s'écrit sur 1024 bits* et qui apparaitra comme *une constante* dans le programme.

Le fichier produit par l'application `POC` sera formé, dans cet ordre, par :

1. les 128 octets du chiffrement RSA de la clef de session AES choisie ;
2. les 16 octets du vecteur d'initialisation choisi ;
3. les octets correspondant au chiffré du fichier indiqué en paramètre.

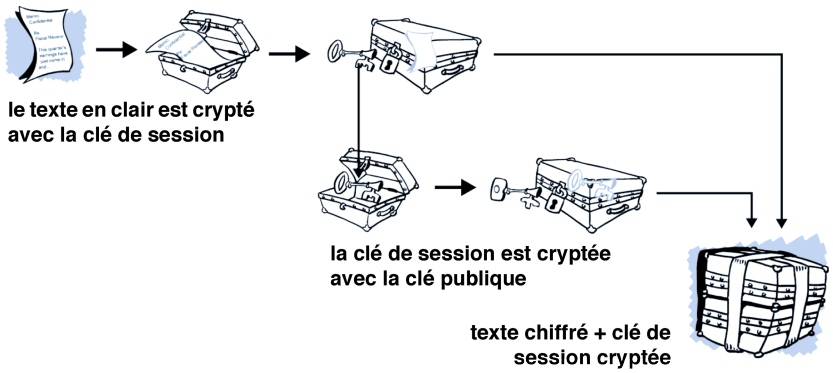

FIGURE 40 – Principe du chiffrement hybride adopté par `POC`

La clef publique utilisée par l'application `POC` est formée de l'exposant $e =$ **`0x44bb...155`**[8] et du module $n =$ **`0x94f2...f21`** de 1024 bits indiqués dans le fichier `clef_publique.txt` de l'archive `POC.zip`.

Exercice G.1 Application POC en JCE Écrire l'application `POC` en Java à l'aide de la JCE.
Sans surprise, le programme `POC` doit, en incorporant des bouts de code vus en cours :

① Choisir de manière aléatoire une clef k AES de 16 octets et un vecteur d'initiatilisation iv de 16 octets ;
② Construire la clef publique RSA qui correspond au module public n et à l'exposant public e indiqués, sous leur forme hexadécimale, dans le fichier `clef_publique.txt` de l'archive `POC.zip` ; puisqu'il s'agit d'une constante, deux objets **`BigInteger`** représentant les entiers n et e fixés seront définis au début de votre programme.
③ Chiffrer en RSA avec bourrage PKCS1 les 16 octets de la clef k à l'aide de cette clef publique ;
④ Écrire les 128 octets du chiffré obtenu dans le fichier résultat ;
⑤ Ajouter ensuite à ce fichier les 16 octets du vecteur d'initialisation ;
⑥ Chiffrer en AES, avec le mode opératoire CBC et le bourrage PKCS5, le fichier indiqué en paramètre à l'aide de la clef AES et du vecteur d'initialisation choisis
⑦ Ajouter dans le fichier résultat les octets correspondant à ce chiffré.

Exercice G.2 Application POC sans JCE Écrire, en C ou en Java, l'application `POC` sans utiliser la JCE mais en utilisant les programmes développés dans les Travaux Pratiques précédents, notamment :
— le TP C sur l'implémentation de l'AES avec bourrage PKCS5 et mode opératoire CBC ;
— l'exercice E.2 sur le bourrage PKCS1 avec RSA.

8. C'est un peu inhabituel pour l'exposant public d'une clef RSA, mais c'est comme ça...

```
$ java POC butokuden.jpg crypt-butokuden.jpg
Taille du fichier à chiffrer: 467796
Taille du fichier à produire: 467952
Clef AES choisie (16 octets):
0xE8C00EFA5FFDE2A73AA61CD4DF616521
Vecteur d'initialisation (16 octets):
0x6E21B4375DB94DEDEBF658BB0039C36E
$ ls -l
total 8304
-rw-r--r--   1 remi  staff     6114 10 fév 10:58 POC.class
-rw-r--r--@  1 remi  staff   467796 10 fév 08:20 butokuden.jpg
-rw-r--r--   1 remi  staff   467952 11 fév 06:34 crypt-butokuden.jpg
```

FIGURE 41 – Exemple d'exécution : chiffrement de `butokuden.jpg` en `crypt-butokuden.jpg`

Exercice G.3 Application dPOC La clef privée d associée à la clef publique utilisée par l'application `POC` vaut $d =$ **`96cf61e6f496c0169825d4c140ea681659b9d42d`** (en hexadécimal) ; ce nombre entier est facilement obtenu comme résumé SHA1 du mot-de-passe « KYOTO » :

```
$ echo -n "KYOTO" | shasum
96cf61e6f496c0169825d4c140ea681659b9d42d
```

Ainsi le journaliste n'a pas besoin de conserver la clef privée : le mot-de-passe « KYOTO » permet de produire la clef privée à utiliser lors du déchiffrement ! Écrire (en Java à l'aide de la JCE) une application `dPOC` qui prend en paramètre un fichier `f` (chiffré avec l'application `POC`) et un mot-de-passe `p` et qui déchiffre le fichier `f` à l'aide du mot-de-passe `p`.

Le programme `dPOC` doit donc :

① Calculer le résumé SHA-1 du mot-de-passe `p` fourni, sous la forme d'un tableau de 20 octets ;

② Construire (à l'aide d'un constructeur adéquat) un objet BigInteger d qui correspond à ce résumé, c'est-à-dire dont l'écriture *en base 256* coïncide avec les octets du résumé ;

③ Construire (à l'aide d'un constructeur adéquat) un objet BigInteger n qui correspond au module public indiqué dans le fichier `clef_publique.txt`, par son écriture en base 16 ;

④ Construire alors la clef privée RSA associée à n et d ;

⑤ Extraire et déchiffrer avec RSA les 128 octets placés au début du fichier `f` afin de produire la clef AES ;

⑥ Extraire ensuite les 16 octets du vecteur d'initialisation ;

⑦ Déchiffrer enfin le reste du fichier `f` à l'aide de la clef AES déchiffrée et du vecteur d'initialisation.

Lorsque `p` est égal à « KYOTO », le programme pourra déchiffrer tous les fichiers produits par l'application `POC`.

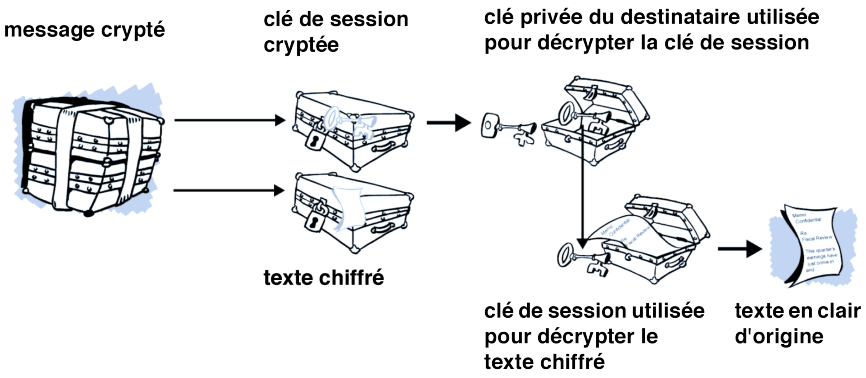

FIGURE 42 – Principe du déchiffrement hybride utilisé par `dPOC`