Rapport de soutenance n° 1

Packup System 4

Packup System 4

MAUBANC "Hyperion" Rémi
RIDEAU "Pokkihju" Cyril
CASIER "Zaihner" Frédéric
FARINAZZO "KernelKiller" Cédric

20 février 2019

Table des matières

Première partie

Introduction

Le projet PS4 est un logiciel de backup de fichiers. Il permettra à terme de sauvegarder ses fichiers et d'en restaurer différents états sauvegardes au préalable. Pour cette première soutenance, nous avons tenté tout d'abord de créer le système de fichiers et de commencer a travailler sur divers algorithmes de compression et de chiffrement.
Concernant l'avancement prévu, nous avons réussi a être dans les temps pour la plupart des parties. Le système de fichier est avancé et les algorithmes de chiffrement sont presque tous implémentés. Il ne reste que les algorithmes AES (une seule erreur à corriger) et d'ElGamal a préparer. Pour la compression, l'algorithme de compression de Huffman est prêt.
Pour cette première soutenance, le projet a donc bien avancé et nous sommes dans les temps que nous avions prévus. Quelques endroits sont en retard mais d'autres en avance, ce qui maintient un équilibre.

Deuxième partie

Le groupe

2.1 Présentation des membres du groupe

2.1.1 MAUBANC Rémi

Elève standard dans cette école, j'ai été accepté dans ce groupe pour mes réalisations extra-scolaire qui est le développement et la mise en production d'un site d'aide aux révisions. Mais également pour des notes pas trop mauvaise en programmation en Sup et pour ma motivation. Ce projet me permettra d'être plus à l'aise avec le language C que je ne maîtrise que peu.

2.1.2 RIDEAU Cyril

Elève en deuxième année à l'EPITA, je suis un grand fan de jeu vidéo, et de programmation. J'ai commencé la programmation en terminale car je souhaitais comprendre comment les programmes que j'appréciais tant fonctionnaient. J'ai donc rejoins l'EPITA. Ce projet m'intéresse car il touche à la gestion I/O [1] et la gestion de fichier. J'aime beaucoup cette partie de la programmation car cela permet de faire beaucoup de choses complexes. J'espère donc que ce projet me permettra de me développer sur ce plan.

2.1.3 CASIER Frédéric

Actuellement en deuxième année à EPITA, je suis un passionné d'art et d'informatique, deux domaines qui occupent la majorité de mon temps libre. J'ai commencé à coder en classe de Terminale, en apprenant les bases du langage Python. Dans le domaine de l'informatique, je suis principalement intéressé par le développement d'interface graphique et le traitement d'image. Toutefois, si j'ai rejoint ce projet, ce n'est pas pour retravailler sur une petite interface, mais pour découvrir d'autres domaines et prendre de l'expérience sur une facette de la programmation où je n'excelle pas.

2.1.4 FARINAZZO Cédric

Passionné d'informatique, ce projet m'intéresse énormément. Le fonctionnement de ce projet me semble complet du point de vue algorithmique. Je suis à l'aise dans la plupart des langages de programmation. Je n'aurai pas trop de problème à réaliser ce projet. Ce projet sera donc pour moi un moyen de découvrir en détail le monde de la cryptographie.

1. Input/Output en francais Entrée/Sortie ici du programme

2.2 Répartition des tâches

Voici la répartition des tâches pour la réalisation du projet :

Tâches	Personnes			
	Rémi	Cyril	Frédéric	Cédric
Système de fichier			X	X
Sauvegarde		X	X	
Compression	X	X		
Chiffrement		X		X
Site web	X			X
Interface graphique	X		X	

Troisième partie

Le projet

3.1 État actuel du projet et prévision pour les prochaines soutenances

Voici un tableau d'avancement des tâches réalisé et de nos prévisions pour les prochaines soutenances :

Tâches	Actuellement	Soutenance	
		n° 2	n° 3
Système de fichier	75%	100%	100%
Sauvegarde	40%	75%	100%
Compression	25%	60%	100%
Chiffrement	50%	80%	100%
Site web	35%	80%	100%
Interface graphique	25%	60%	100%

3.2 Sauvegarde

3.2.1 Système de fichier

Pour cette soutenance, nous avons commencé le système de fichiers. Nous avons commencé par créer l'arbre représentant l'arborescence des fichiers. Ceci est nécessaire pour la restauration des fichiers mais aussi pour leur sauvegarde et la sauvegarde des différents états. Pour construire cet arbre, il nous a fallu parcourir toute l'arborescence d'un chemin donne.
Pour cela, nous avons choisi d'utiliser la bibliothèque standard et *dirent*. Parcourir un fichier ne s'est pas avéré très complique au début, nous avons même réussi a faire une fonction qui permet de copier-coller des dossiers très facilement lors de nos essais. Cependant, lors de la construction de l'arbre, plusieurs problèmes se sont posés. Tout d'abord, il faut gérer les cas de récursion (dossier dans un dossier) mais aussi les cas terminaux : fichiers ou dossiers vides.
Pour toutes ces raisons, nous avons décidé d'implémenter ceci de manière récursive et de faire plusieurs fonctions. Nous avons donc une fonction qui crée l'arbre si c'est un fichier et une autre si c'est un dossier. Cependant, si c'est un dossier, il a fallu gérer le cas particulier du dossier vide. Pour cela nous avons une condition spéciale dans la fonction.

Un autre problème est arrive ensuite. Sous Linux tout est fichier, y compris les dossiers. Nous avons donc du comprendre comment distinguer un dossier d'un autre fichier pour gérer nos appels correctement. Heureusement, les librairies standard contiennent déjà des éléments pour nous aider a gérer ceci.
Ainsi, dans la fonction principale, nous faisons une première vérification du type de fichier. Si celui-ci est un fichier "simple", nous appelons la sous-fonction qui gère les fichiers et sinon, nous appelons la fonction qui gère les dossiers qui elle-même appelle récursivement la première ou elle-même.
Le second problème majeur que nous avons rencontre est la présence des dossiers "." et ".." dans chaque répertoire. Ces deux dossiers représentent respectivement le dossier actuel et le dossier parent. Le problème de ces fichiers est qu'ils sont toujours présents et que si on appelle récursivement dessus, on s'expose a une récursion infinie, retournant en boucle dans le répertoire actuel ou dans les dossiers parents jusqu'au root, posant ainsi de potentiels problèmes d'autorisation.
Afin d'éviter ce problème nous sommes forces de vérifier chaque appel pour ne pas appeler sur ces dossiers, sous peine de crash.

```
1   struct meta_data
2   {
3       char *path;
4       struct stat fs;
5   }meta_data;
6
7   struct meta_tree
8   {
9   struct meta_data *data;
10  struct meta_tree *son;
11  struct meta_tree *sibling;
12  }meta_tree;
13
```

Ci-dessus se trouvent les structures utilisées pour stocker l'arborescence de fichiers. Pour le moment, les données stockées sont simples mais il est possible d'en rajouter de plus complexes, notamment la version etc...

Une fois que l'arbre a été construit, il a fallu le sauvegarder et pour cela nous avons décidé de tout gérer nous même. La structure contenant des pointeurs, il est inutile de sauvegarder ceci puisque ce qu'ils représentent sera perdu entre temps. Il est plus intéressant de sauvegarder ce qu'ils représentent. Pour commencer, la sauvegarde a été découpée en sauvegarde des données, c'est a dire chemin et statistiques et en sauvegarde récursive.
Pour sauvegarder les données, nous avons choisi de sauvegarder d'abord la longueur du chemin puis celui-ci et enfin les statistiques du fichier. Ceci a été fait afin de faciliter la restauration.
Pour sauvegarder le reste de l'arbre, il a fallu sauvegarder un élément permettant de savoir si le noeud a un frère ou un fils afin de savoir si le prochain noeud sauvegarde est lie a l'actuel. Pour cela, nous avons décidé de sauvegarder ces deux informations dans un byte en créant des macros et en utilisant des opérations bitwise. Cela nous permet de simplifier la restauration.

3.2.2 Restauration

Ayant précédemment sauvegarder l'arbre, il faut le restaurer en mémoire. Pour cela, nous effectuons l'opération inverse de la sauvegarde. Tout d'abord, il faut lire la longueur du chemin qui suit. Ensuite, le chemin est lu et les statistiques aussi. Une fois tout cela effectue, la structure meta_data est recrée. C'est ensuite au tour du byte stockant les informations sur les fils du noeud d'être lu, ce qui permet de contrôler les appels de restauration récursifs.
Le reste de la restauration n'est plus constitué que d'appels récursifs jusqu'à la fin du fichier et la restauration de l'intégralité de l'arbre.

3.3 Compression

Avant de chiffrer les données et de réduire l'espace occupé, il est essentiel de les compresser auparavant. Comme exprimer dans le cahier des charges, ce sont les algorithmes de compression Huffman et LZ78 qui ont sélectionnés pour ce projet.

Pour cette première soutenance, l'avancement est un peu plus faible que celui annoncé dans le cahier des charges. La raison ? Des lacunes. En effet, la vitesse de réalisation des algorithmes était beaucoup plus faible que prévu, dû un manque de compréhension sur les pointeurs du langage C.

3.3.1 Huffman

Pour cette soutenance, l'algorithme de compression de Huffman est opérationnel. En revanche, les contre-temps n'ont pas permis de mettre au point la décompression. La compression de Huffman se base sur la répétition des caractères pour la compression. Plus un caractère est présent dans la chaîne de départ plus il sera codé sur un nombre de bit réduit dans la chaîne compressée. En premier, nous construisons une liste avec tous les caractères présent et leur nombre d'apparition associé. Ce tableau doit être ensuite trié pour être utilisé. La liste est triée dans l'ordre décroissant. Avec la liste obtenue, nous construisons un arbre binaire où chaque feuille contient un caractère de la précédente liste. Les noeud interne ne contiennent pas de valeur utile. Dans la théorie ils contiennent ϵ, dans notre projet, ils contiendront le caractère NULL. La construction de l'arbre se passe de la sorte (pour une chaîne contenant au moins trois caractères différents) :

1. Les deux premiers éléments sont placés en feuille et sont fils d'un unique noeud père. Ce dernier devient le fils gauche du noeud racine de l'arbre final.
2. Un noeud contenant ϵ est ajouté en fils droit de la racine. Et nous plaçons un pointeur sur ce dernier
3. Les deux éléments suivant, s'ils existent sont ajoutés en fils et feuille d'un unique noeud père. Ce dernier devient le fils gauche du noeud pointé
4. Un noeud contenant ϵ est ajouté en fils droit du noeud pointé. Le pointeur est déplacé sur son fils droit.
5. Quand il n'y a plus d'élément, l'avant dernier fils droit de l'arbre final est remplacé par le fils gauche de ce dernier.

Les étapes 3 et 4 sont répétées récursivement jusqu'à qu'il n'y ait plus d'élément dans la liste. Il reste à construire la table de codage qui donne la représentation binaire de chaque caractère de la chaîne. La méthode est simple : il suffit de faire un parcours profondeur de l'arbre en ajoutant en préfixe un 0 si on passe sur un fils gauche, ou un 1 si on passe sur un fils droit. Dès qu'on arrive sur une feuille on ajoute à la table de codage, le contenu de la chaîne préfixe et la valeur de la

clé de la feuille. Il a été remarqué que nous pouvons utiliser la table de codage pour reconstruire l'arbre. Ainsi, nous la recyclons pour la compression de l'arbre de Huffman. Cette manipulation nous évite de refaire plusieurs parcours de l'arbre et d'allouer une autre liste, ce qui optimise significativement le temps et les ressources nécessaire au programme de compression. Pour encoder la chaîne en entrée, il nous suffit de remplacer tout les caractères par leur équivalent dans la table de codage. Elle sera alors en binaire ; la dernière étape consiste à convertir le binaire en ASCII et passer d'une liste chaînée dynamique à une chaîne de caractère statique. Pour l'encodage de l'arbre, il suffit de convertir en binaire tous les caractères différents de 0 ou de 1 en binaire et reconvertir en ASCII l'ensemble. Pendant la conversion en binaire, la longueur des chaînes de caractères des données et/ou de l'arbre ne sont pas nécessairement un multiple de 8, c'est pourquoi il est nécessaire de rajouter des 0 pour compléter. Cette manipulation nous oblige a transmettre le nombre de bits rajouté pour l'alignement. A des fins de présentation, un ratio est affiché sur le terminal, son calcul est simple :

$$ratio = \frac{longueur(donnéescompressées) + longueur(arbrecompressé)}{longueur(donnéesentrées)} * 100$$

Même si la décompression n'est pas opérationnelle, les structures de données ont déjà été créée et une partie méthodes théorique en pseudo-algo ont été écrites (pour le décodage de l'arbre).

3.4 Chiffrement

Pour cette première soutenance, nous avons implémentes quatre algorithmes de chiffrement.

3.4.1 Rotn

Nous avons tout d'abord implémenté l'algorithme Rotn. Cette algorithme de chiffrement par substitution utilise une clé qui est un simple entier. C'est une généralisation du Chiffre de César. Il consiste à remplacer systématiquement dans le message clair une lettre donnée (dans notre cas un code ascii) de l'alphabet par un signe donné en fonction de la clé. C'est donc une simple addition pour chaque charactère du code ascii du caractère avec la clé. Le déchiffrement est donc la différence du caractère encodé avec la clé.

3.4.2 Vigenère

Le chiffre de Vigenère est un système de chiffrement par substitution, mais une même caractère du message clair peut, suivant sa position dans celui-ci, être remplacée par des lettres différentes en fonction de la clé qui, contrairement à un système de chiffrement tel que Rotn ou le chiffre de César, est d'une chaine de caractère. Cette méthode résiste ainsi à l'analyse de fréquences. Il suit le même fonctionnement que Rotn à la différence que l'on utilise les caractères de la clé pour chiffrer le message.

3.4.3 RSA

Nous avons aussi été implémenté l'algorithme de chiffrement asymétrique RSA. Pour la réalisation de cet algorithme, nous avons utilisé la bibliothèque (The GNU Multiple Precision Arithmetic Library `https://gmplib.org/`) pour éviter tous overflow dû au fonctionnement d'RSA. Cette bibliothèque nous met à disposition le type de donnes mpz_t, qui nous permet de travailler avec de grands nombres sans craindre un overflow. Notre algorithme génère donc la clé privée et la clé publique à partir de 2 nombres premiers.

```
1  struct RSA_publickey {
2      mpz_t *n;
3      mpz_t *e;
4  } RSA_publickey;
5
6  struct RSA_privatekey {
7      mpz_t *n;
8      mpz_t *d;
9  } RSA_privatekey;
10
```

```
1  struct RSA_publickey *RSA_gen_public_key(mpz_t p, mpz_t q);
2
3  struct RSA_privatekey *RSA_gen_private_key(mpz_t p, mpz_t q, struct
   RSA_publickey *public);
4
```

s

Pour chiffrer, nous avons juste besoin d'appeler la fonction RSA_encode avec la clé publique et une chaine de caractère. Cette fonction nous retourne la chaine encodée sous la forme d'un tableau de mpz_t.

La fonction RSA_decode permet de déchiffrer le tableau de mpz_t avec la clé privée et nous retourne la chaine de caractère décodée.

3.4.4 AES

AES est un algorithme de chiffrement symétrique par bloc de 16 caractères (sous la forme de matrice). Dans la version AES 128, que nous avons implémenté, il est constitué d'un algorithme d'expansion de la clé qui nous permet d'obtenir 10 autres clés devant être utilisé pour le chiffrement/déchiffrement et de 4 algorithmes (AddRoundKey, Subbytes, Shiftrows, Mixcolumns) utilisé dans l'algorithme de chiffrement (pour le déchiffrement, l'inverse de ces 4 algorithmes est utilisé). AES 128 utilise donc ces différents algorithmes pendant les 10 tours nécessaires à son fonctionnement. L'algorithme de déchiffrement reprend le même principe.

Nous avons donc créé une nouvelle structure AES_matrix pour simplifier la création des algorithmes.

```
1    struct AES_matrix {
2        size_t rowsLenght;
3        size_t colsLenght;
4        uint8_t **data;
5    } AES_matrix;
6
```

Après plusieurs tests sur chacun des algorithmes, tous semblaient fonctionnel mise à part l'algorithme Mixcolumns qui ne modifie pas correctement la matrice, entraînant un mauvais chiffrement. Nous réglerons ce problème pour la prochaine soutenance.

Pour la prochaine soutenance, nous prévoyons de résoudre le problème de l'algorithme Mixcolumns d'AES et de commencer l'implémentation du cryptosystème d'ElGamal.

3.5 L'interface graphique

Un logiciel de sauvegarde peut être compliquer à utiliser pour l'utilisateur. C'est pour cela qu'une interface graphique est plus que nécessaire.
Bien qu'elle ne doit pas être excessivement complète et qu'elle peut reste simple d'utilisation, l'interface devra tout de même fournir tous les outils nécessaire pour compresser et chiffrer les document désires.
De plus incorporer une barre de progression a notre interface pour informer l'utilisateur du temps restant est aussi implémentable, pour avoir une interface agréable a regarder et ne pas être vide durant l'exécution du programme. Les objectifs actuellement fixe pour une interface sont donc les suivant : Une interface simple et efficace réalisé grâce à la librairie GTK-3, donner accès a tous les outils nécessaire à l'utilisateur pour effectuer des sauvegardes, permettre à l'utilisateur de sélectionner les fichiers et document à compresser/chiffrer et créer une interface appréciable à regarder, pouvant informer l'avancement du programme après son exécution.

L'interface actuelle est composé d'un explorateur de fichier et de boutons non fonctionnels pour le moment. La création de cette interface fut possible à l'aide du logiciel Glade, permettant la construction d'un squelette pour une interface, par la suite relié à un code en C pour rendre le tout fonctionnelle.

Pour la suite, la priorité sera de lier les parties des autres membres du groupe à l'interface pour pouvoir avoir accès à tous les tests nécessaires via cette interface. Dans la finalité, l'interface sera plus épurée, et ne sera plus composé d'une seul fenêtre, pour rendre le tout plus interactif et facile d'utilisation.

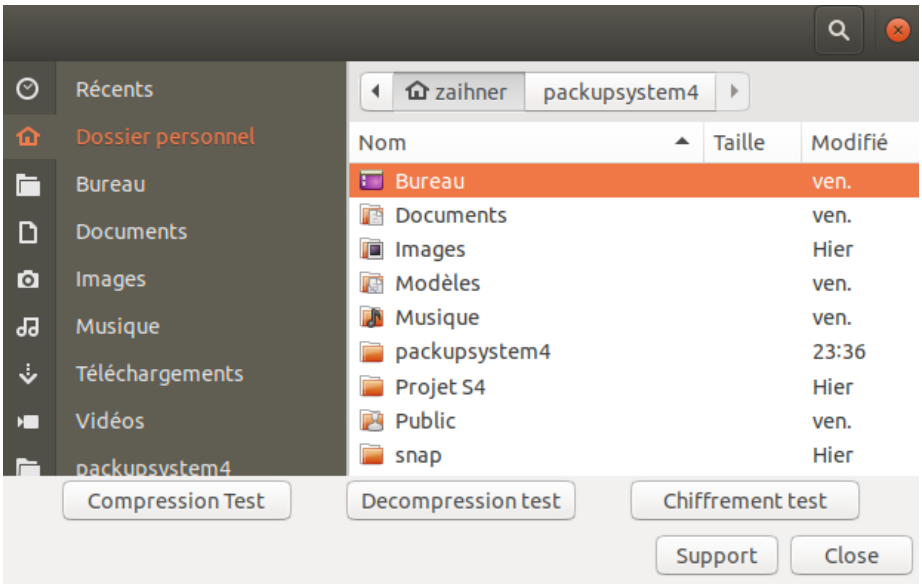

FIGURE 3.1 – Interface actuel

3.6 Site web

Le site web est disponible à l'adresse suivante : `https://packup.hyperion.tf/`.
Il est actuellement hébergé sur un VPS [1] d'OVH déjà utilisé par Hyperion pour herbergé plusieurs sites web. Ce serveur était idéal car il possède une configuration opérationnelle. Notre choix pour les parties de back-end et de front-end ont été très largement influencé par les infrastructure web déjà en place. Ainsi, nous avons choisit le framework Django `https://www.djangoproject.com/` pour le site web qui fait appel au serveur d'application uwsgi et au reverse proxy nginx. Pour le style et l'affichage, nous utilisons le framework Materialize (`https://materializecss.com/`) afin de vous présenter un site web clair et responsive.

Pour la prochaine soutenance, nous prévoyons de créer le système d'authentification.

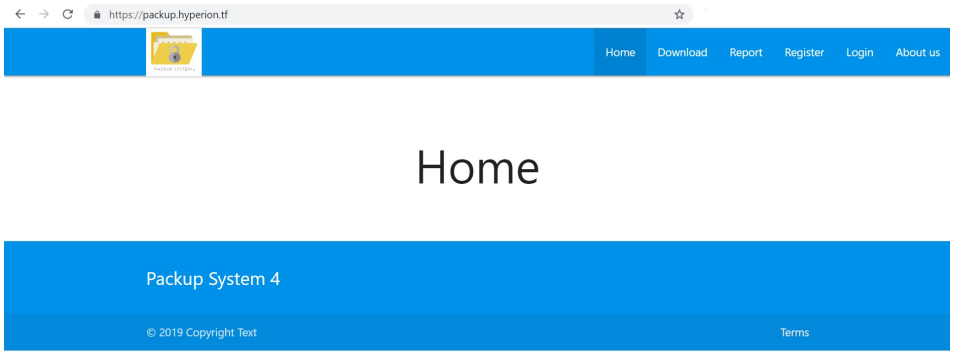

FIGURE 3.2 – Site web actuel

1. VPS (Virtual Private Server) = Serveur privé virtuel

3.7 Avancement prévu pour les prochaines soutenances

Voici un tableau d'avancement des tâches prévu pour les prochaines soutenances :

Tâches	Soutenance		
	n° 1	n° 2	n° 3
Système de fichier	75%	100%	100%
Sauvegarde	50%	75%	100%
Compression	30%	60%	100%
Chiffrement	30%	80%	100%
Site web	50%	80%	100%
Interface graphique	25%	60%	100%

Quatrième partie

Conclusion

Pour conclure, nous sommes dans les temps pour le projet, le système de fichier est fonctionnel bien qu'il soit incomplet et le système de sauvegarde qui dépend de ce dernier est par conséquent dans les temps aussi. La compression est bien avancée et l'algorithme d'Huffman est implémenté a moitié, il ne reste qu'a régler des problèmes pour la décompression. Le chiffrement est la partie avec le plus de contenu fonctionnel a cause de la simplicité des premiers algorithmes implémentés. En effet, Rotn et Vigenere sont des algorithmes très simples et qui ne requièrent pas beaucoup de travail. Cependant, il y a aussi des algorithmes très complexes termines dans cette partie notamment AES et RSA.
Sur le côté utilisateur, l'interface graphique est presque prête et il ne reste qu'à la lier aux autres parties en permettant à l'utilisateur de choisir quel fichier sauvegarder, où et si ils doivent être chiffres ou non.
Le projet est donc en bonne voie pour être réalisé a temps et fonctionnel.