Rapport de stage

Guillaume Hautot

26 août 2025

FIGURE 1

Projet soutenu par le ministère de la Culture

MINISTÈRE
DE LA CULTURE
*Liberté
Égalité
Fraternité*

Table des matières

1 Introduction

Ce stage de fin d'année a été pour moi une occasion de découvrir l'environnement réel d'un laboratoire d'informatique et d'appliquer les connaissances accumulées depuis ma licence. Ce stage m'a également permis de me familiariser avec la forensique numérique.

1.1 Présentation du GREYC

J'ai été accueilli au sein du laboratoire du GREYC, le *groupe de recherche en informatique, image automatique, et instrumentation de Caen*. Cet établissement est dédié à l'étude d'une variété de domaines en lien avec l'informatique.

Les travaux de recherche se concentrent sur l'algorithmique, la cryptographie, l'instrumentation, la science des données et l'IA.

Les différentes équipes

Le GREYC est divisé en différentes équipes travaillant de manière isolées :

- L'équipe SAFE : Sécurité, Architecture, Forensique, biomÉtrie
- L'équipe MAD : Modèles, Agents, Décision
- L'équipe AMACC : Algorithmes, Modèles de calcul, Aléa, Combinatoire, Cryptographie, Complexité
- L'équipe Image : chargé des recherches sur le traitement d'image
- L'équipe CODAG : Contraintes, Ontologies, Données, Annotations, Graphes
- L'équipe Électronique : chargé des recherches sur l'instrumentation

1.2 Présentation de l'IMEC

L'*institut mémoire de l'édition contemporaine* (IMEC) est un organisme fondé en 1988 collectant et conservant des fonds d'archives de différentes maisons d'édition.

La majorité des fonds possédés par cet institut est présentée sous format papier. L'intégralité des archives conservées s'étend sur 33 ans, couvrant des documents produits entre 1986 et 2019.

Parmi la minorité de fonds numériques y figurent des images disques d'anciennes versions des distributions *Apple* de l'époque : les distributions *Macintosh classiques*.

1.3 Objectifs du projet IANEC

Le projet d'*investigation d'archives numériques d'écrivains contemporains* (IANEC) a été réalisé par l'équipe SAFE en collaboration avec l'IMEC. Celui-ci a pour but d'élaborer différentes solutions permettant de rendre possible ou plus simple le travail des archivistes de l'IMEC sur leurs fonds numériques.

Plus concrètement, il nous a été demandé dans un premier temps d'automatiser l'analyse des images disque, pour parvenir à détecter le système d'exploitation installé (si il y en a un), les différentes applications présentes, et distinguer les fichiers système des fichiers liés à utilisateur.
Dans un second temps, il a fallu faciliter le travail des archivistes en leur apportant des outils de classification automatique, permettant de donner un indice sur le contenu et la communicabilité de ces documents.

Fonds fourni par l'IMEC

Pour nous aider dans ce projet, l'IMEC nous a fourni différents fonds numériques pour tester nos idées, afin de mettre au point des solutions. Ces fonds comportaient des copies bit à bit de disques, disquettes, et CD-ROM, ainsi que des fichiers extraits de ces différents périphériques de stockage.

Parmi les images disques, les systèmes installés étaient essentiellement des systèmes *Macintosh Classique*, s'étalant de la version 6 à la version 9, et il y avait également des versions de *Mac OS X*. En effet, certains fonds pouvant être anciens ("anciens" dans le langage d'un informaticien), il a fallu développer des solutions spécifiques aux technologies informatiques de l'époque.

Division des tâches

Deux sous-groupes se sont formés pour répondre aux attentes du projet. Le premier groupe était uniquement composé d'étudiants en cybersécurité à l'université de Caen, constitué de Titouan Le Bret, Matthias David, et moi-même. C'est à nous qu'a été attribuée la première tâche d'analyse des disques.

Le second groupe, était formé par Mohammed Salah Aissahoui, Dihia Slimana qui étaient tout deux également en master de cybersécurité, et Hamza

Ouazzani Chahdi, étudiant à l'ENSI Caen. Ce groupe était chargé de développer les solutions de classification automatique.

2 Travaux réalisés

2.1 Solution de distinction des fichiers systèmes et utilisateurs

Problématique

L'objectif principal de notre groupe était de créer différents outils pour aider les archivistes à localiser les fichiers dignes de leurs intérêts. Ceux-ci comprenaient en autre, des e-mails, des photos, des documents textes formatés ou non, mais également des polices d'écriture (qui peuvent avoir une valeur sémantique dans du texte formaté).

Il nous a semblé bien plus simple de réfléchir par soustraction : si on identifie les fichiers systèmes et les fichiers des applications, il nous restera uniquement ce qui intéresse nos archivistes, sauf peut-être pour le cas des polices d'écriture.

Il nous restait donc à trouver un moyen d'identifier ces fichiers systèmes et fichiers d'applications (que l'on regroupera par le terme de *fichiers techniques*). Pour ce faire, nous avons eu l'idée de collecter les différents fichiers depuis des images disques de différentes versions des systèmes Macintosh classique nouvellement installés. Ces disques ne contenaient donc aucune donnée d'utilisateurs. Il nous a paru naturel de hacher ces fichiers, et de collecter uniquement leurs empreintes. Cela nous a permis de stocker uniquement l'empreinte, réduisant la quantité de données à conserver sur le disque.

Base d'empreintes du NIST

Ces *bases d'empreintes* sont une pratique courante et essentielle en forensique numérique. En effet, elles permettent d'identifier des fichiers sans avoir à les stocker entièrement, ni à les communiquer, les rendant très utiles pour la détection de contenus illégaux dans les systèmes de fichiers.

Il s'avère que le *national institute of standards and technology* (NIST) a un projet de construction d'une base de données contenant des empreintes de fichiers techniques, nommé le projet NSRL (pour *national software reference library*).

Ce projet a produit une multitude de jeux de données de références (RDS), notamment une collection *legacy* pour les empreintes de fichiers techniques antérieurs à 2015, provenant de plusieurs systèmes d'exploitation. Ces *hashsets* sont entièrement téléversés sur le site du NIST tous les ans, avec une mise à jour incrémentielle disponible tous les 3 mois pour chacun d'eux. Nous avons décidé de récupérer une de ces collections, et de réutiliser le même *schéma* que cette dernière pour les autres bases de données que nous créerons.

Les RDS sont disponibles en deux variétés différentes, ayant deux schémas différents : une version complète qui fournit une plus grande quantité de métadonnées sur les fichiers techniques, et une version minimale plus légère.

La version complète n'a pas été retenue en raison de sa taille (40 Go sous format compréssé) et de la trop grande richesse de ses tables (beaucoup d'attributs par table). En effet, si nous souhaitons faire nos propres bases de données, il nous a semblé essentiel de faire en sorte que les archivistes puissent incrémenter les bases de données d'eux-mêmes sans avoir à fournir d'informations superflues.

Ce schéma contient cinq tables devant être remplies manuellement et six tables de jointure (voir 1). Cette grande quantité de tables de jointures permet de diminuer la redondance des données dans les différentes tables. En revanche, elles sont plus complexe à utiliser, particulièrement si nous devons développer des outils pour manipuler les *BDDs*. Nous avons donc choisi d'utiliser le schéma minimal (voir 2).

Les deux schémas stockent plusieurs empreintes différentes pour chaque fichier, provenant de plusieurs fonctions de hachage différentes : MD5, *SHA-256*, *SHA-1*, et *CRC-32*. Dans la table minimale, ces informations sont présentes dans la table *FILE*. Utiliser plusieurs fonctions de hachage permet d'éliminer toute possibilité de collision sur des fichiers qui seraient différents. Cette multitude de types d'empreintes permet également d'identifier des fichiers dont on aurait uniquement pour information un seul type d'empreinte parmi les quatre.

Pour la version minimale, la table *PKG* contient les données de différentes applications. Chaque ligne de la table FILE référence un PKG. Chaque application référence un système d'exploitation de la table *OS*. Les fichiers des systèmes d'exploitation ne sont pas directement référencés vers la table OS. Il est donc nécessaire de créer une ligne dans PKG, pour faire office d'intermédiaire, afin d'ajouter des fichiers appartenant à un système d'exploitation.

La table *VERSION*, présente dans les deux schémas, contient une seule ligne donnant des informations sur la base de données.

De *Sqlite* à *PostgreSQL*

Après quelques recherches à l'intérieur de cette base de données, nous nous sommes aperçu que peu de fichiers techniques des Macintosh classique y étaient présents. La majorité des fichiers techniques provenaient de systèmes *Windows* et *Mac OS X*.

Il était donc, comme prévu, nécessaire de construire notre propre base de données d'empreintes, sur le même schéma que celles du NIST.

Les hashsets fournis par le NIST sont sous format *Sqlite*. C'est une technologie reconnue pour sa simplicité et son efficacité, sans gestion de privilèges, avec une base de données par fichiers. Malgré cela, c'est un outil qui représente également des désavantages, notamment l'impossibilité d'accéder aux bases de

données Sqlite par le réseau, pas de support natif pour l'indexation des colonnes par table de hachage, une nécessité de gérer proprement les fichiers des bases de données.

Nous avons donc opté pour l'utilisation de *PostgreSQL*, nous permettant :

- de travailler en réseau, qui présente plusieurs intérêts :
  - pouvoir ajouter des données de manière collaborative, afin d'accélérer le travail ;
  - stocker les données sur une unique machine, que chacun d'entre nous possède une copie de la base ;
  - faire en sorte que cette base de données puisse être déployée dans le réseau privé de l'IMEC ;
- de remplacer les index *B-Tree* de SQLite par des index en table de hachage, particulièrement intéressant pour des données dont l'ordre n'a pas d'importance comme des empreintes numériques ;
- d'ajouter des extensions si nécessaire ;

En faisant le calcul, il y a $n = 166\,560\,293$ fichiers différents dans *RDS legacy minimal* et si on analyse une image disque avec $k = 2000$ fichiers, alors : sur Sqlite, une analyse disque prend environs $k \cdot \log_2(n) \approx 54623$ recherches dans la base de données ; sur PostgreSQL, elle prend $k = 2000$ recherches. On passe d'une complexité en temps de $\Theta(k \cdot \ln(n))$ à $\Theta(k)$.

On peut faire le même calcul pour l'ajout de données à la base, qui elle aussi passe d'une complexité en temps de $\Theta(k \cdot \ln(n))$ à $\Theta(k)$ avec k le nombre de fichiers à ajouter et n la quantité de fichiers déjà présents.

Pour transformer la base du NIST du format Sqlite vers le format de PostgreSQL, j'ai utilisé un outil nommé *pgloader*. Cet outil permet de sélectionner les différents objets que l'on souhaite transférer vers la base *Postgres*, me permettant d'exclure les *index Sqlite*.

Pour recréer l'index sur PostgreSQL, j'ai utilisé la commande

```
CREATE INDEX ON file USING hash (md5);
```

Étonnamment, l'indexation par table de hachage est plus légère en stockage que celle par arbre binaire. Lors du premier essai de la copie de la BDD Sqlite vers *Postgres*, j'avais laissé l'indexation en *B-Tree*, qui a eu pour conséquence de remplir la quantité de stockage qui m'était accordé sur ma machine.

Pour la colonne d'indexation, j'avais le choix entre les colonnes MD5, SHA-256, SHA-1, et CRC-32, ou alors d'une combinaison entre plusieurs de ces colonnes.

CRC-32 étant une fonction de hachage avec de petites empreintes (32 bits), cela semblait évident qu'il y aurait des collisions : il y a 166 560 293 lignes dans la table FILE, et $2^{32} = 4\,294\,967\,296$ empreintes possibles, ce qui me paraît bien trop faible si l'on souhaite avoir un index sans collisions, en gardant à l'esprit le paradoxe des anniversaires.

Utiliser la fonctionnalité d'*indexation sur plusieurs colonnes* d'un autre côté, aurait pu résoudre définitivement ce problème, avec une taille en entrée de 576 bits en tout. Mais cela nous a paru trop long à hacher si on devait répéter cette opération un grand nombre de fois. De plus, il est plus flexible d'utiliser un index sur une unique colonne, pour rendre possible la recherche sur celle-ci.

Au final, j'ai décidé d'indexer sur la colonne MD5 parce que celle-ci est assez grande pour avoir une très faible probabilité de collisions, mais pas *trop* grande, si jamais on devait avoir besoin de performance.

La meilleure solution aurait été d'exploiter directement l'uniformité de la colonne MD5 pour y dériver sa position dans la table, mais PostgreSQL n'offre pas cette possibilité. Par ailleurs, les empreintes stockées par le NIST sont sous format *ASCII* héxadécimal, ce qui fait doubler la taille réelle des empreintes.

À postériori, j'ai découvert que PostgreSQL utilisait une fonction de hachage avec 32 bits de sortie, augmentant la quantité de collisions dans les buckets.

Sécuriser les bases de données

Avant d'exposer le service PostgreSQL au réseau interne du GREYC, il a fallu identifier les différentes menaces et configurer les différentes connexions que celle-ci peut accepter. Il était tout de même préférable de prendre des précautions, surtout dans des environnements collaboratifs. Nous n'avions pas d'information sur la taille réelle du réseau, et donc sur la surface d'attaque qu'elle représentait, mais nous avons estimé qu'un accès par mot de passe serait suffisant.

La menace la plus importante restait à mes yeux l'erreur humaine, surtout pour de la manipulation de bases de données. J'ai donc décidé de mettre en place une stratégie de défense en profondeur, et de créer différents rôles sur l'*instance PostgreSQL*, liés à chacune des manières d'utiliser la base de données.

Il y a donc le *rôle postgres*, présent par défaut, qui a tous les droits sur tous les schémas et toutes les options, de toutes les bases de l'instance. Concrètement, c'est équivalent à l'utilisateur *root* sur les systèmes d'exploitation de type *Unix* (à quelques nuances près). Son usage est avant tout pour la création de bases de données et l'ajout d'extensions.

J'ai créé le rôle *rw_user*, qui a accès à tous les schémas de toutes les bases en lecture-écriture. C'est le rôle qui est censé être utilisé pour ajouter des données à la base. Je lui ai d'abord laisser les privilèges `INSERT`, `UPDATE` et `DELETE`, pour faire du débogage au début. Je lui ai ensuite retiré `UPDATE` et `DELETE`. Il aurait peut-être été préférable de faire un rôle distinct *debug* pour permettre l'altération et la suppression des lignes, et permettre a rw_user de garder la fonction prévue à son origine.

Enfin, le dernier rôle est *ro_user*, qui n'a que le droit d'accès en lecture aux schémas. C'est ce rôle qui sera utilisé pour détecter les différents fichiers techniques présents sur les images disque.

L'accès au service PostgreSQL étant utilisé seulement par moi, Matthias, et Titouan, nous avons décidé que ces trois rôles auraient le même mot de passe, afin de ne pas modifier la variable `PG_PASSWORD` contenu dans l'environnement *Bash*, qui est lu automatiquement par le client PostgreSQL pour se connecter.

L'accès physique non autorisé était également une possibilité à prendre en compte. Sachant que les données étaient conservées sur le disque de ma machine, et que l'accès du bureau était contrôlé, une intrusion aurait été difficile à mettre en place, et peu digne d'intérêt pour un attaquant.

J'ai fait en sorte d'installer la base de données sur un disque virtuel différent de mon disque virtuel de travail, afin de rendre la base plus facile à déplacer, ou à changer de machine virtuelle. À la place du système de fichiers *ext4*, j'ai utilisé *XFS*. C'est un système de fichiers qui est recommandé pour les bases de données.

À ce stade, nous avions 3 bases de données dans l'instance : la base de données *hashdb* qui contiendrait les empreintes que nous allions ajouter, créé par le GREYC ; *hashdb_test* utilisé pour les essais des différents clients que nous développions parallèlement ; et la base *rds_legacy_minimal*, qui contenait la base de données de NIST.

La table *rds_legacy_minimal* étant censé résider dans l'instance sans être modifiée. J'ai donc activé le paramètre `default_transaction_read_only=on`, qui protège l'accès en écriture spécifiquement pour cette base de données, même pour le rôle *postgres*.

Ajouter des données aux bases

La pierre angulaire du travail était posée, nous avions une instance PostgreSQL, un schéma réutilisable et différentes bases de données à renseigner ou utiliser.

Pour pouvoir pleinement utiliser ces bases de données, nous avions à créer des outils pour :

- ajouter des informations aux différentes BDDs de manière automatisée, et créer des nouvelles BDDs sur le même schéma que RDS ;
- utiliser les informations dans une combinaison de BDDs, afin de permettre l'identification des fichiers techniques, et de fournir des rapports synthétiques ou analytiques.

Nous avons d'abord construit l'outils *hash_match_manager*, facilitant l'écriture de nouvelles données dans les bases.

L'utilisation de *Python 3* a eu de plusieurs avantages, notamment la librairie standard contenant beaucoup de *modules* simples et puissants, mais également car le python est un langage qui nous est familier.

Ce programme a été conçu à la fois pour une utilisation en *ligne de commandes*, et pour une utilisation en *interface graphique*, pour les utilisateurs occasionnels. Nous avons hésité à faire une *interface en ligne de texte* à la place

d'une interface en ligne de commandes, mais nous avons gardé le CLI pour potentiellement intégrer les fonctionnalités du client à des scripts Bash.

La partie graphique inclut quatre formulaires, pour remplir chacune des tables du schéma (en excluant la table *VERSION*). Le formulaire de la table FILE permet de choisir un chemin du système de fichiers, afin d'ajouter tous les sous-fichiers du répertoire de manière récursive ; ou d'ajouter uniquement un seul fichier si l'entité du chemin spécifié est un fichier. En réussissant à extraire les données d'une image disque, ou en la montant, cela nous permet donc de capturer les empreintes de tous les fichiers présents, en ayant seulement à spécifier la racine.

On a ajouté une fonctionnalité similaire, permettant à l'utilisateur de choisir une image disque, de monter automatiquement celle-ci, et d'ajouter des fichiers récursivement depuis le point de montage. Le montage se fait par l'intermédiaire d'un *loop device*, permettant d'interpréter un fichier comme un périphérique de bloc. Malgré cela, cette méthode n'est pas recommandée, car certains systèmes de fichiers montés sous *Linux* n'exposent pas toutes leurs informations au système de fichiers virtuel. Nous avons utilisé le *module Python TKinter* pour construire cette interface graphique car c'est un module présent par défaut.

L'interface en ligne de commande ajoute des données avec les différents arguments fournis. Elle contient les mêmes fonctionnalités que la version graphique. *Python* fourni le module *argparse*, qui permet d'analyser les arguments passés en lignes de commande et créer des interpréteurs personnalisés. Cela a permis de faire plusieurs sous-commandes :

- `hashdb gui` pour lancer l'interface graphique
- `hashdb connect` pour se connecter avec le client psql à une session *configurée*
- `hashdb create { file | pkg | os | mfg } ...` pour ajouter différentes données à la base

C'est un module puissant qui nous a également permit de formater automatiquement l'usage de la commande, et de générer une option `--help`.

Pour configurer la connexion du programme à la base de données, nous avons utilisé un fichier de configuration regroupant les informations nécessaires. Le fichier *connection_cfg.yaml* contient l'adresse IP du réseau du service, le nom de la base de données ciblé, et le nom du rôle utilisé pour y accéder.

Pour rendre le programme plus facilement portable, nous avons choisi de mettre ce fichier de configuration directement dans la racine du projet au lieu d'utiliser un chemin spécifique à chaque systèmes d'exploitation (par exemple `~/.config` pour les systèmes de type Linux, ou `C:\Users\Utilisateur\AppData\Roaming` pour les distributions Windows)

Nous avons également utilisé le module *psycopg2*, qui lui n'est pas inclus dans la librairie standard. Ce module permet de communiquer avec l'instance PostgreSQL avec des requêtes préparées.

Du point de vue du schéma de la base de données, les empreintes des fichiers techniques ne sont pas référencées par les systèmes d'exploitation, on ne sait donc pas quels fichiers appartiennent à quels OS. Il est possible de voir cela en remarquant qu'il n'y a pas de clé étrangère vers la table OS dans la table FILE.

Nous avons donc fait en sorte de créer un *package* pour chaque système d'exploitation ajouté, de tel sorte que la table OS puisse être utilisée comme une extension de la table PKG. Cela permet donc que chaque système soit référencé par la table des empreintes. Pour identifier ces packages générés automatiquement lors de la création d'un système d'exploitation, nous leur avons donné un nom particulier : `{Nom de l'OS} {Version de l'OS} (Vierge)`.

Pour ajouter des fichiers d'application à la base de données, il fallait prendre en compte les fichiers systèmes déjà ajoutés. En effet, il fallait s'assurer, pour chacun des fichiers de l'application, de les ajouter uniquement s'ils ne sont pas présents dans le système d'exploitation. C'est une contrainte importante, car on avait prévu de mesurer le taux d'incidence de chacune des applications trouvées. La mesure de ce taux d'incidence permettant d'identifier quelles applications se trouvaient sur le disque de l'écrivain.

Pour ce faire, il a fallu adapter les requêtes d'ajout à la table FILE, si le package est une application :

```
-- Requete A
INSERT INTO file (sha256, sha1, md5, crc32, file_name, file_size,
    package_id)
SELECT %(sha256), %(sha1), %(md5), %(crc32), %(file_name), %(file_size),
    %(package_id)
WHERE (%(sha256), %(sha1), %(md5), %(crc32), %(file_name), %(file_size))
NOT IN (
  -- Requete B
  SELECT (f.sha256, f.sha1, f.md5, f.crc32)
  FROM file f
    INNER JOIN pkg p ON f.package_id = p.package_id
    INNER JOIN os o ON p.operating_system_id = o.operating_system_id
  WHERE p.name LIKE '%(Vierge)'
    AND operating_system_id = (
        -- Requete C
        SELECT operating_system_id FROM pkg WHERE package_id =
            %(package_id)
      )
);
```

La requête A ajoute les données du fichier si les informations données ne sont pas dans la requête B.
La requête B sélectionne tous les fichiers de tous les systèmes d'exploitation installés.
La requête C permet de filtrer la requête B afin de récupérer les fichiers du système d'exploitation sur lequel est installé l'application.

Trouver les informations à ajouter

Par la suite, nous avons dû agrandir la base de données *hashdb*. Pour ce faire, nous avons utilisé différents émulateurs afin d'installer les différentes versions des systèmes d'exploitation *Macintosh classique*.

Les systèmes *Macintosh classique*, c'est-à-dire antérieurs à Mac OS X, ont subi des évolutions radicales au cours de leurs successions.

Les architectures se sont succédées, d'abord *Motorola 68K* présent depuis le début des systèmes Mac, puis *PowerPC* à partir de Mac 7, et enfin les ISA Intel 32 et 64 bits.

C'est un aspect important à prendre en compte, car un jeu d'instruction différent signifie des programmes binaires différents, et donc plus de fichiers techniques à répertorier. De même pour les différentes langues pour chaque applications et systèmes.

Nous avons dû faire des choix dans les différentes données. On a ainsi priorisé, les systèmes et applications en langue française, sur architecture *ppc*.

Nous avons principalement utilisé *SheepShaver* pour émuler l'architecture *PowerPC* et *Mini vMac* pour *Motorola 68K*. Ces outils étaient également utilisés par les archivistes de l'IMEC. Les deux programmes émulent les périphériques de stockage par des images disques présentes sur le système de fichiers. Il faut également leur fournir une *ROM* Macintosh *Old World* ou *New World*. Ces ROMs contiennent le firmware de la carte mère, nécessaire au démarrage des systèmes.

Plusieurs communautés en ligne fournissent ces ROMs, mais également des disques et disquettes d'installations des MacOS, ainsi que des programmes. Parmi les plus utilisés, il y a notamment *macintoshrepository.org* et *archive.org*.

Nous avons donc ajouté plusieurs systèmes MacOS sur des architectures différentes sur la base de données. Nous avons également ajouté plusieurs éditeurs de texte formatés, ainsi que d'autres programmes utilitaires.

Analyser des disques avec la base de données

Pour analyser les disques, les archivistes créent des copies numériques de celles-ci. Sur cette copie, ils utilisent le programme *Autopsy*, qui est un outil libre utilisé pour la forensique numérique. Cet outil est codé en *Java*, et est basé sur *SleuthKit*, qui est une panoplie d'utilitaires forensique.

Autopsy peut être étendu grâce à un système de *plugins*. Ces plugins sont codés en *Java*. Ils peuvent aussi être écrit en *Python* et interprété par *Jython*, qui est un interpréteur *Python* écrit en *Java*.

Nous sommes donc partis sur l'idée de créer un plugin permettant de communiquer avec le schéma de nos bases de données. Celui-ci remplirait plusieurs objectifs :

1. se connecter à une ou plusieurs bases de données ;

2. récupérer les empreintes des fichiers d'un disque, et les envoyés aux bases ;
3. générer des rapports sur les empreintes retrouvés dans les bases de données, afin d'en déduire les systèmes et applications installés.

Nous avons d'abord essayé d'utiliser *Python* pour ce plugin. L'inconvénient principal est que *Jython* interprète uniquement le *Python2*. Nous avons donc préféré programmer ce plugin dans le même langage qu'a été programmé *Autopsy*, afin de faciliter l'intégration.

J'ai aidé Titouan au début de ce projet, mais je me suis concentré sur d'autres travaux par la suite.

2.2 Lire les systèmes de fichiers d'Apple sous Linux

Je me suis intéressé à ouvrir les différents systèmes de fichiers des MacOS. L'objectif est de pouvoir récupérer les informations présentes sur images disques sans avoir la nécessité de passer par un émulateur. Cela permettrait de traiter de grandes quantités de disques sans nécessiter d'interventions manuelles.

Les archivistes de l'IMEC ont le plus souvent à faire à des systèmes de fichiers HFS et HFS+. Il s'avère qu'il est déjà possible d'ouvrir ces systèmes de fichiers sous Linux nativement. Seulement, Linux n'affiche pas toutes les données véritablement présentes sur les disques ou images disques.

Apple a, à la création de *MFS*, son premier système de fichiers, créé un concept de *fourche d'informations* (*fork* en anglais). C'est un segment de données, similaire aux données que l'on trouve lorsque l'on ouvre un fichier texte sur Windows ou Linux. À la différence près, qu'il existe deux fourches par fichier :

- Une fourche de données (data fork), qui contient les données du fichier en lui-même (les caractères d'un fichier texte non formaté, les couleurs des pixels d'une *bitmap*, ...).
- Une fourche de ressources (resource fork), qui contient des métadonnées utilisées par le fichier (les polices d'écriture d'un fichier texte formaté, l'icône du fichier, ...).

On a donc un chemin de fichier qui est en fait lié à deux sources d'informations (voir 3). C'est un concept particulier, car nous avons l'habitude de concevoir un fichier étant directement rattaché à ses données.

Pour lire les fourches de données contenues dans les système de fichiers HFS, les archivistes m'ont confié utiliser *hfsutils*, qui est une suite d'outils reposant sur la librairie *libhfs*.

Ces outils sont des équivalents des utilitaires principaux de *GNU*, adaptés pour des images HFS. Par exemple, `hmount` permet de simuler le montage d'un disque ; `hls` permet de lister les différents fichiers en lui donnant un argument un chemin MacOS ; `hcopy` permet de copier un fichier du disque vers le système de fichiers local (voir 6).

Ces outils présentent des inconvénients. D'abord, il n'y a pas de programme `hcat`, pour lire une fourche d'un fichier. Ainsi, si je souhaite lire les données d'un fichier de mon disque HFS sous format hexadécimal : je dois le copier avec `hcopy`, puis le lire avec `xxd` ou `hexdump`. Il n'y a pas de programme `hxxd` fourni.

Si je souhaite faire un `grep` récursif sur tous les fichiers de mon disque : je dois lister les fichiers avec `hls`, copier chacun des fichiers, puis faire un grep, et enfin faire de même sur les autres dossiers récursivement avec chaque programme.

Il est totalement possible de faire cela dans un script Bash, mais cela les rend plus complexes à écrire. C'est également frustrant si l'objectif n'est que d'extraire la moitié des données. Il serait plus simple d' utiliser les différents utilitaires déjà présents sur le système, de la même manière que sur les autres fichiers du système.

Une méthode qui pourrait résoudre ce problème serait de réimplémenter le montage de *HFS* comme on le désire, c'est-à-dire en y incluant les données manquantes. Cela nous permettrait également d'éviter de copier les fichiers que l'on souhaite lire, et de les interpréter directement depuis le support.

J'avais entendu parler auparavent d'un module du *noyau Linux* permettant aux utilisateurs, même non privilégiés, de monter des systèmes de fichiers en tant qu'utilisateur. Ce module s'appelle *FUSE* (pour *filesystem in userland*).

Le *VFS*, le système de fichiers virtuel, est l'arborescence des répertoires et des fichiers, celui-ci est géré par le noyau du système d'exploitation. Chaque systèmes de fichiers concrets, comme *ext4*, *XFS*, ou *NTFS*, implémentent la manière de devenir une branche (ou la racine) du *VFS* (voir 5). Ces systèmes de fichiers, que l'on va abréger par *FS*, gèrent leurs propres manière de lire et écrire dans les fichiers et les dossiers, sur demande du *VFS*.

Ces implémentations sont codées dans le noyau, et sont donc gérées par le système. C'est également le cas pour le module FUSE, mais au lieu de définir un moyen d'accéder aux *entités* d'un *FS*, il laisse un programme utilisateur définir ces fonctionnalités. Le travail principal du module FUSE est d'agir comme un relai sécurisé entre le *VFS* et le programme utilisateur, si jamais celui-ci contient des failles, est mal utilisé, ou est volontairement codé a des fins malveillantes.

On comprend ici le terme *entité* comme étant soit un fichier, soit un dossier.

À ma surprise, il n'existe pas de projet d'implémentation de HFS standard en tant que programme FUSE, ainsi que pour MFS, le premier *FS* d'*Apple*. Il existe déjà un autre projet pour monter les systèmes de fichiers HFS+ et ses dérivés (HFSJ sa version journalisée, et HFSX sa version sensible à la case), qui se nomme *hfsfuse*.

J'ai donc essayé de combler ce manque en créant à mon tour un *driver* FUSE pour HFS. En conjonction avec la librairie FUSE, j'ai réutilisé la librairie libhfs utilisée par les outils de hfsutils pour lire dans les différentes *entités du système*, de manière totalement agnostique quant au fonctionnement réel de HFS.

Deux choix d'implémentations s'offraient à moi. La première est de réutiliser les programmes compilés de la suite hfsutils directement dans mon programme. À chaque demandes du *VFS*, le programme devait :

1. Décoder l'action demandée par FUSE (et donc du *VFS*).
2. Éxecuter le programme hfsutils correspondant à l'action.
3. Récupérer la sortie de celui-ci.
4. Formater les données pour les renvoyer à FUSE.

La lecture des fichiers aurait pu ce faire avec un tuyau. Au lieu de copier directement sur le système de fichiers avec `hcopy <chemin Mac> <chemin Linux>`, on lance `hcopy <chemin Mac> /dev/stdout`, afin de remplacer la destination par la sortie standard du sous-processus `hcopy`. Cette sortie est donc récupérable de la même manière que toute les autres sorties des differents programmes.

Cette première stratégie me parraissait la plus simple à mettre en place, même dans un langage haut niveau comme *Python*. L'inconvénient principal est la nécessité de lancer un nouveau processus *à chaques actions envoyées par FUSE*, m'assurant une performance désastreuse. En cas d'un `find` sur la racine, ou d'un *grep -r*, la quantité de sous-processus à créer et à attendre sera bien trop élevée. Également, seules les fonctionnalités présentes dans les programmes peuvent être réutilisées.

La seconde stratégie aurait été de réutiliser directement les fonctions de libhfs, me permettant de lire les images sans intermédiaire (voir 7). Cette méthode est plus complexe à mettre en œuvre, mais j'ai tout de même choisi de partir sur cette idée.

Elle offre en revanche des avantages non négligeables, notamment la possibilité de me servir de toute la librairie hfsutils, et de meilleures performances.

Le second choix a été le langage de programmation. Je suis d'abord parti sur du *Python*, car c'est un langage simple avec une grande librairie. Malgrés l'existence de *bindings* pour *libfuse* en *Python*, libhfs était quant à lui uniquement disponible en langage C. J'ai d'abord utilisé *cffi* pour importer les fonctions et structures de libhfs vers Python. Cela requiert de compiler libhfs avec le compilateur de *cffi*, pour produire un module *Python*.

Le problème est que certains `#define` s étaient importants à garder dans le code de libhfs, que cffi ne gardait pour Python. J'ai du implémenter une manière de les récupérer :

```
def get_defines(header: Path):
  result = sp.run(["gcc", "-dM", "-E", header], capture_output=True,
      text=True).stdout
  defines = {}
  for line in result.splitlines():
    if line.startswith('#define'):
```

```
    parts = line.split(maxsplit=2)
    if len(parts) == 3: # Certains defines sont sans valeur, comme par
        exemple les gardes fous des headers
      [_, name, value] = parts
      defines[name] = value
return defines
```

Malgrés cela, il restait le cas des *macros fonctionnels* qui ne pouvaient plus être interprétés comme fonction.

L'utilisation du *C* dans du *Python* a rendu le déboggage bien plus compliqué, notamment pour les erreurs de segmentation. Il fallait également transformer les différents types du *C* vers le *Python*, et inversement, rendant le code plus verbeux.

J'ai donc abandonné l'idée de coder mon programme dans ce langage. J'ai hésité à le faire en *C++* qui est un langage avec lequel j'ai acquis une certaine expérience. C'est un langage qui est rétrocompatible avec le *C*. J'ai tout de même opté pour du *C* pour m'assurer de l'intégration.

Pour le gestionnaire de projet, je trouvais que la solution la plus simple était d'utiliser *CMake*. Ce gestionnaire de projet m'a permis d'ajouter des tests unitaires, en tant que sous-projet.

Je n'ai pas utilisé la source officiel de libhfs, mais un *fork* de hfsutils. Ce *fork* a été créé par un utilisateur nommé *targetdisk*. libhfs y est présent dans un sous-répertoire de ce projet. J'ai utilisé ce fork spécifiquement car il résout des problèmes de compilations. libhfs étant un projet ancien, il utilise des programmes anciens pour gérer les macros. Ces programmes existent toujours, mais ils ont beaucoup évolués.

Par la suite, j'ai créé un *fork* de ce dernier, pour modifier quelques fonctions et l'adapter à mon usage. En effet, le *fork* de *targetdisk* compilait tout, mais n'installait pas la librairie libhfs ni ses *en-têtes*. La librairie libhfs était liée statiquement aux programmes hfsutils. Je n'ai pas souhaité changer le mode d'édition des liens vers du dynamique.

La librairie libhfs contient des fonctions qui sont très similaires à celles de l'*API* de FUSE :

Fonction libhfs	Interface FUSE	Intéragit avec ...
`hfs_mount`	`init`	Système de fichiers
`hfs_umount`	`destroy`	
`hfs_stat`	`getattr`	Entités
`hfs_readlink`	`readlink`	Liens symboliques
`hfs_opendir`	`opendir`	Répertoires
`hfs_readdir`	`readdir`	
`hfs_closedir`	`releasedir`	
`hfs_open`	`open`	fichiers
`hfs_read`	`read`	
`hfs_close`	`release`	

La librairie FUSE fournit un type `fuse_operations`. Cette structure contient des attributs qui ont pour type des pointeurs de fonctions. Ces pointeurs de fonctions référencent l'implémentation dans mon code *C*. Une fois cette structure remplie, on la passe à FUSE par la fonction `fuse_main`.

Dans `main.c` :

```c
#include <hfuse.h>
int main(int argc, char *argv[]) {
    if(argc <= 2) {
        fprintf(stderr, "Usage: %s <image> <mountpoint> [fuse
            args...]\n", argv[0]);
        return 1;
    }
      /* ... */
    return fuse_main(args.argc, args.argv, &hfuse_operations, (void*)
        context);
}
```

Dans `hfuse.c` :

```c
const struct fuse_operations hfuse_operations = {
    .init         = hfuse_init,
    .destroy      = hfuse_destroy,
    .getattr      = hfuse_getattr,
      /* ... */
    .release      = hfuse_release,
}
```

Étant donné que ce programme a été créé dans l'objectif de lire des images disque d'archives, je n'ai pas implémenté l'écriture dans les fichiers, ainsi que la création de nouveau fichiers dans les répertoires.

Il existe d'autres fonctions implémentables à l'intérieur de `fuse_operations`, 40 au total. Je n'ai implémenté que les fonctions les plus importantes. Les fonctions restantes ont des implémentations par défaut.

Les chemins

Les chemins des systèmes MacOS classique sont différents des chemins Linux. Le séparateur de chemin n'est pas le caractère `/`, mais le caractère `:`. C'est également les séparateurs de chemins de libhfs. Il a donc fallu traduire les chemins demandés par le *VFS*, vers des chemins lisibles par libhfs. À noter que les chemins donnés par le *VFS* ont pour racine le point de montage, et non pas la racine du *VFS*.

À l'inverse, les *noms d'entités* renvoyés lors des appels à `readdir` ne devaient pas contenir de caractère `/`. Par *noms d'entités*, nous entendons les noms des fichiers et des dossiers, sans leurs chemins.

Une première solution est d'intervertir les `/` et `:`. Dans ce cas, pas besoin de faire de fonction très complexes, seulement une copie du chemin donné par le *VFS*, et une boucle sur chaque caractère pour faire l'interversion.

Je ne suis pas parti sur cette idée car je n'étais pas sûr que ce soit la seule contrainte des chemins de HFS. J'ai décidé de faire un encodage ressemblant à l'encodage URL. J'ai transformé les `/` présents dans les noms de fichiers et dossiers HFS en `%2F`. Si il se trouve qu'un autre caractère n'est pas accépté par HFS, je peux encoder celui-ci de la même manière.

Il fallait également représenter les différentes *fourches d'informations*. La première approche qui m'était venue à l'esprit est d'écrire un *fichier virtuel* par *fourche*. Si on a un fichier `:dossier:mon_fichier.doc`, alors, lors de la lecture de chaque entité du répertoire `/mnt/dossier` avec `readdir`, je liste les *fichiers virtuels* `mon_fichier.doc.data` pour contenir la *fourche de données* et `mon_fichier.doc.rsrc` pour y contenir la *fourche de ressources*. Cela a pour conséquence de créer beaucoup de *fichiers virtuels* dans les répertoires.

La seconde approche est de créer un *répertoire virtuel* `.rsrc` pour chaque répertoire de la partition. Dans ces répertoires, j'y ajoute les mêmes noms de fichiers que les fichiers du répertoire parent. Il y aura donc `/mnt/dossier/mon_fichier.doc` contenant les données et `/mnt/dossier/.rsrc/mon_fichier.doc` contenant les ressources. C'est également cette architecture qui est utilisée par SheepShaver lorsque l'on copie des fichiers depuis une machine émulée vers l'hôte. J'ai décidé de réutiliser cette structure ci (voir 4).

Seulement avec le chemin demandé par le *VFS*, il est nécessaire de pouvoir reconstituer toutes les informations permettant de savoir quelle fourche de quel fichier est demandée par l'utilisateur. On ne peut pas garder d'informations à mesure que l'on traverse le système de fichiers. C'est-à-dire que si j'attribue des métadonnées à un répertoire `.rsrc`, je ne peux pas réutiliser ces métadonnées pour lire le fichier `mon_fichier.doc` à l'intérieur de celui-ci. Chaque chemin est indépendant. Une fois la bonne *fourche* donnée à `hfs_setfork`, on peut écrire sur le *buffer* donné par `fuse_operations.read` en paramètres avec `hfs_read`.

Voici le processus complet de la transformation d'un chemin de ma structure,

vers un chemin HFS :

1. Un chemin est donné par le *VFS* : `/Bureau/Ecrits/.rsrc/fichier_avec_%2F.pdf`
2. On récupère quelle *fork* a été choisie, en supprimant le répertoire `.rsrc` du chemin s'il est présent : `/Bureau/Ecrits/fichiers_avec_%2F.pdf`
3. On décode le chemin pour libhfs : `:Bureau:Ecrits:fichier_avec_/.pdf`

Déboggage

Pour valider le comportement attendu de ces fonctions, j'ai créé des *tests unitaires*. J'ai créé ces tests avec *CTest*. Pour cela, j'ai fait en sorte de compiler tout mon projet en tant que librairie, sauf pour la fonction *main*. Ainsi, je peux lier la librairie soit au programme réel dans le `main.c`, soit dans mes différents tests unitaires.

Ces tests unitaires sont à l'intérieur du répertoire `test` à la racine du projet *hfuse*. C'est simplement un sous-projet *CMake* qui est appelé par le projet parent *hfuse*.

Pour chasser les fuites mémoires, engendrées par l'usage de `malloc`, j'ai utilisé *valgrind* avec l'option `-leak-check=full`. J'ai dû l'utiliser en *root* car *valgrind* analyse aussi la partie module de FUSE, nécessitant des privilèges. J'ai hésité à faire les transformations de chaînes de caractères avec `alloca`, mais cette méthode ne fonctionne pas sur tout les compilateurs.

Pour les problèmes liés à la sémantique du code, j'ai créé des fonctions de déboggage. Ces fonctions me permettent d'afficher plus facilement certaines valeurs, notamment les chaines de caractères qui peuvent être vides. J'ai également utilisé l'option `-d` en paramètre de mon programme, permettant de récupérer des informations de déboggage que l'API FUSEaffiche dans la sortie standard.

Implémentation de `readdir`

L'API FUSE permet d'implémenter la fontion `fuse_operations.readdir` de deux manières. Voici la signature de cette fonction, avec uniquement les paramètres que j'utilise :

- `void *buf` contenant les informations accumulées au file des appels de `filler` ;
- `fuse_fill_dir_t filler`, qui est une fonction définie par FUSE pour ajouter des entités dans `buf` ;
- `off_t offset`, un entier permettant de définir le décalage de l'élément actuel ;
- `struct fuse_file_info *fi`, contenant des métainformations remplies par FUSE, et contenant également un *handler* `fi->fh`, qui est un pointeur vers une structure que j'ai définie.

La premier *mode* lit tout le répertoire en un seul appel à l'implémentation de `readdir`. On appelle plusieurs fois `filler` sur un seul appel de `readdir`. Pour ce faire, le paramètre `offset` lors des appels à `filler` doivent être mis à `0`. Ce premier mode est plus simple à implémenter que le suivant, mais il est plus long à exécuter. Si le `buffer` est dépassé, FUSE rappelle une seconde fois `readdir` avec un *buffer* ayant une taille adaptée.

Le second mode permet d'ajouter des données en s'assurant que le buffer n'est jamais rempli. En incrémentant l'offset à chaque appels de `readdir`, FUSE va rappeler cette dernière avec un nouveau `buffer`. Il faut donc s'assurer que `readdir` finisse juste après avoir appelé `filler`. J'ai privilégié cette méthode car la fonction `hfs_readdir` est implémentée de manière similaire, dans libhfs.

Je n'ai pas eu besoin d'utiliser l'argument `const char* path`, dans mon implémentation de `readdir`. J'ai seulement eu à réutiliser ce que j'ai déjà calculé par la fonction `opendir`. Cette fonction prend en paramètre `struct fuse_file_info *fi`, regroupant des informations générées par FUSE, ainsi qu'un `fi->fh` utilisé en tant que pointeur vers une structure que j'ai définie. Voici la définition de cette structure :

```
struct hfuse_handler_s {
    void* structure;      /* Pointeur vers une structure 'hfsdir' ou
        'hfsfile' de libhfs */
    char* macpath;        /* Chemin libhfs (avec les séparateurs ':') */
    dirtype_t dirtype;    /* Type de la fourche (data ou resource) */
    enttype_t enttype;    /* Type d'entité (fichier ou répertoire)
        m'indiquant si la structure pointée est un 'hfsdir' ou un
        'hfsfile' */
};
```

Cette structure est donc initalisée et remplie lors de l'appel à `opendir`, utilisée par `readdir`, et détruite par `releasedir`. J'ai utilisé le même procédé pour les fichiers, avec `open`, `read`, et `release` respectivement.

Les *handlers* de répertoires et de fichiers ont des fonctionnalités similaires : l'initialisation, l'ouverture, et la fermeture. Pour cette raison, j'ai fait en sorte que cette structure soit utilisée pour les deux types d'entités possibles. Pour la simplicité, je n'ai pas fait de méthodes virtuelles, j'ai défini un comportement en fonction du type d'entité, avec des branchements conditionnels (*switch-case*).

L'implémentation de la lecture est laissée aux fonctions `read` et `readdir`, ces deux fonctions dépendent uniquement du type de *fourche*. Pour `readdir`, si la fourche de resource est choisie (on se trouve dans un `.rsrc`), alors il faut faire en sorte de ne pas lister les répertoires, c'est pour cela que l'implémentation dépend du type de *fourche*.

La meilleure solution aurait sûrement été de faire en sorte que `hfuse_handler` soit l'équivalent d'une *classe abstraite* en *Java*. Les deux *classes abstraites filles*

`hfuse_file_handler` et `hfuse_dir_handler` ajoutant une méthode `read` chacune de signature différentes, et implémentant `init`, `open` et `release`. Enfin, avec quatre classes concrêtes différentes, implémenter chaque cas possibles.

Pour les liens symboliques, j'ai implémenté une version minimale permettant simplement de les distinguer du reste des fichiers. Dans mon système de fichiers, ils pointent vers la chaîne de caractère vide. Les références des liens symboliques dans les partitions HFS sont stockés dans les *forks de ressources*. Je ne l'ai pas implémenté car je n'ai trouvé aucune fonction de libhfs permettant de faire cela. Également, je considère que ce n'est pas un choix intéressant de les implémenter si l'on souhaite faire de la forensique.

Améliorations possibles

Je suis fière du résultat de cet outil. Il est possible d'ouvrir toutes les fourches de chaque fichier. Malgrès tout, il reste des améliorations qui peuvent être apportées.

D'abord, les MacOS classiques n'utilisent pas le caractère *LF* comme sous Linux, mais *CR*. Il aurait peut-être été intéressant d'ajouter une option afin de les transformer lors de l'appel à `read`, pour une lecture plus simple. En revanche cette fonctionnalité n'est pas intéressante pour de la forensique, elle n'était donc pas une priorité.

J'ai découvert tardivement le système d'*attributs étendu* de Linux (*xattrs*). J'ai eu pour idée d'ajouter ces attributs à chaque fichier virtuel créé pour différencier les fourches. C'est intéressant car si jamais on a des difficultés à trier les fourches par une expression régulière, on pourrait plutôt se fier à un xattr. En revanche un tel système requiert des appels supplémentaires sur le système de fichiers virtuel.

Également, le multithreading natif de FUSE m'a posé des problèmes. Je l'ai donc désactivé dans le `main`, avec l'option `-s` donné à la fonction `fuse_main`. Je crois que ces problèmes sont liés au fait que libhfs est une librairie *stateful* (par exemple avec `hfs_readdir`). Il est sûrement possible de contourner ce problème avec les fonctions `lock` ou `flock` implémentables.

De plus, au niveau des *fourches de ressources*, il aurait peut être été préférable de ne pas afficher celles qui sont vides. Lors de la lecture d'un répertoire `.rsrc`, le programme lit tout le répertoire parent et liste seulement les fichiers.

Problèmes lié au C

À postériori, le C n'a peut-être pas été le langage le plus intéressant. Ce qui m'a le plus géné est l'abscence de *namespace*, j'aurais aimé pouvoir écrire mes

fonctions dans mon propre espace de noms, pour les rendre plus lisibles. Je pense que j'aurai pu faire un bon usage des classes, en factorisant plus facilement les *handlers* comme vu précédement.

3 Conclusion

Ce stage de fin d'année m'a fait découvrir l'environnement de travail d'un laboratoire spécialisé en informatique. J'ai développé mes connaissainces sur le fonctionnement du monde de la recherche.

En fin de stage, j'ai visité l'IMEC, située à l'abbaye d'Ardennes. Nous y avons présenté ce que nous avions construit.

Les travaux réalisés leur ont été envoyé. Matthias a créé un programme permettant de gérer des profiles d'émulation, Titouan s'est charché du client permettant de trouver des hashs sur un disque. Dihia et Mohammed ont créé un programme permettant de résumer des documents utilisateurs basé sur IA.

Nous avons également créé une machine virtuelle sous *Ubuntu*, dans lequel nous y avions installé tous les programmes que nous avons créé au cours de ces quatre mois, ainsi que la base de données. Les archivistes de l'IMEC avaient l'air satisfait des travaux rélisés.

J'ai approfondi ma maîtrise des langages C, Python, et Bash et découvert certains mécanismes du système Linux comme les *loop devices*. J'ai acquis des connaissances techniques sur plusieurs sujets, notamment la gestion et la sécurisation de bases de données, ainsi que le développement de système de fichiers avec FUSE.

Ce stage m'a conforté dans mon intérêt pour les thématiques liées à la sécurité, et m'a permis de découvrir la forensique numérique. Cette période en laboratoire a été ma première expérience concrète de recherche appliquée, et constitue une étape précieuse pour la suite de mon parcours académique et professionnel.

A Annexe

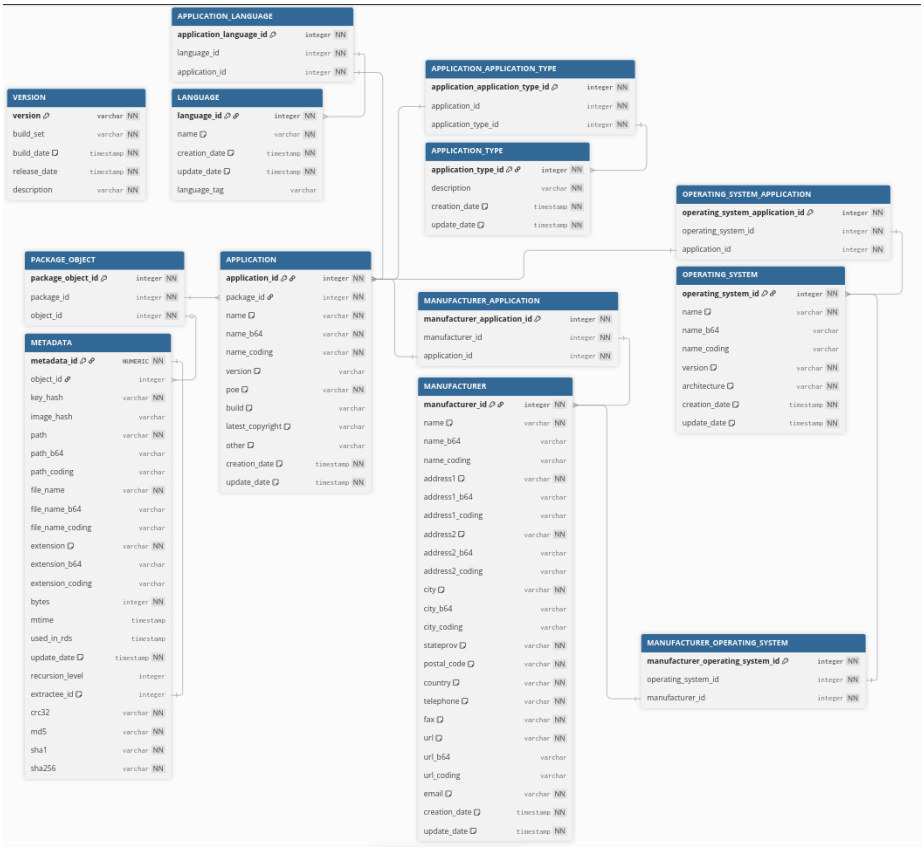

FIGURE 1 – Schéma relationnel de la version complète de RDS legacy

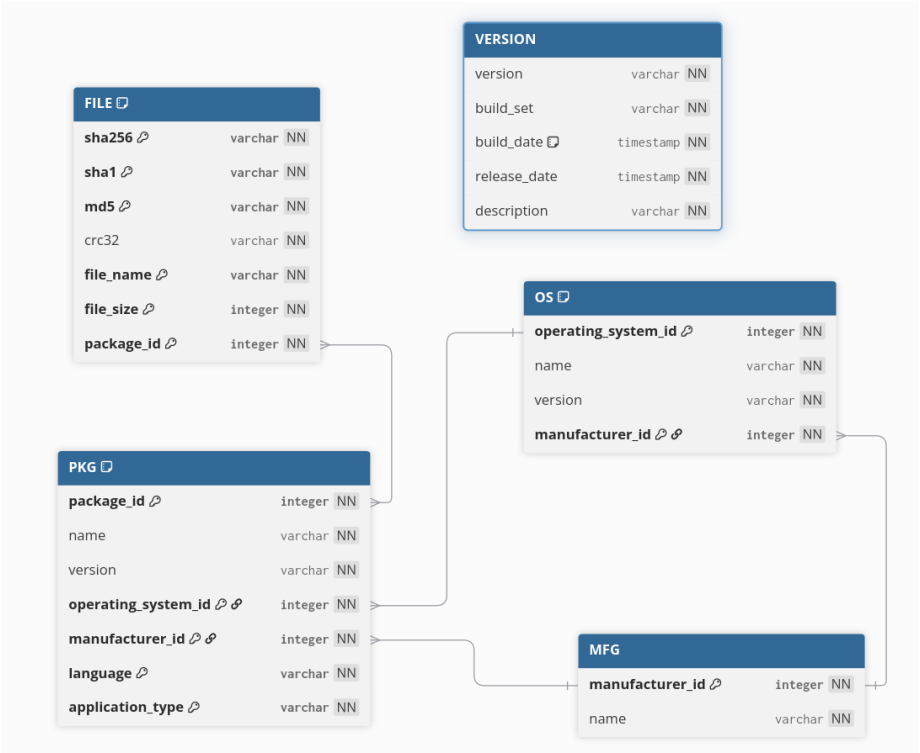

FIGURE 2 – Schéma relationnel de la version minimale de RDS legacy

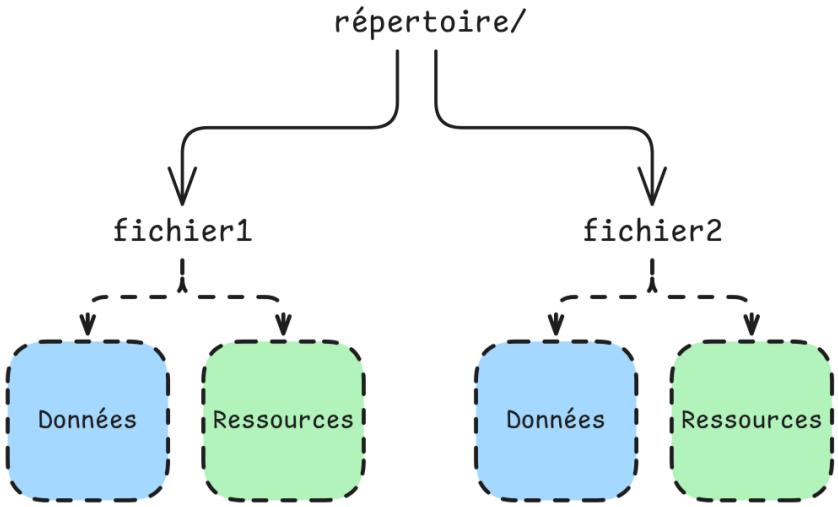

FIGURE 3 – Fonctionnement des fourches dans le système de fichiers HFS.

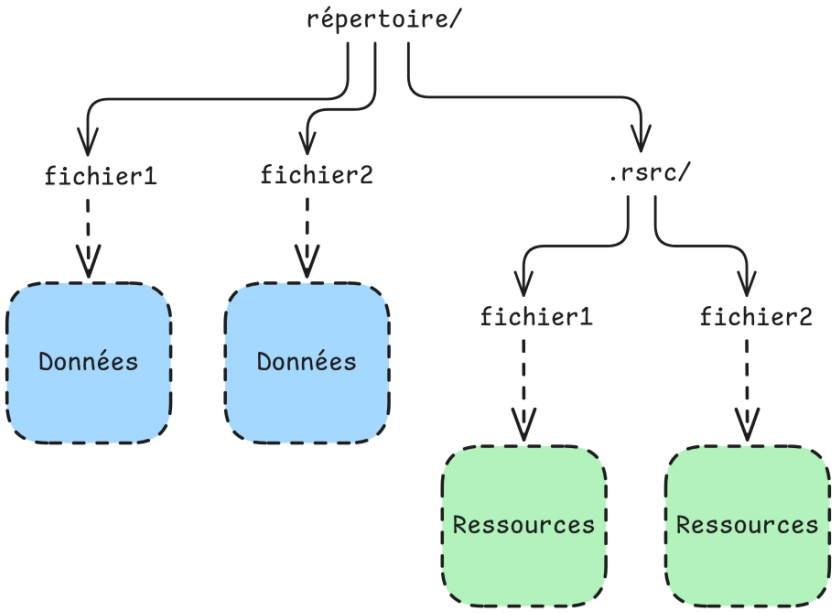

FIGURE 4 – Structure des répertoires de SheepShaver et hfuse

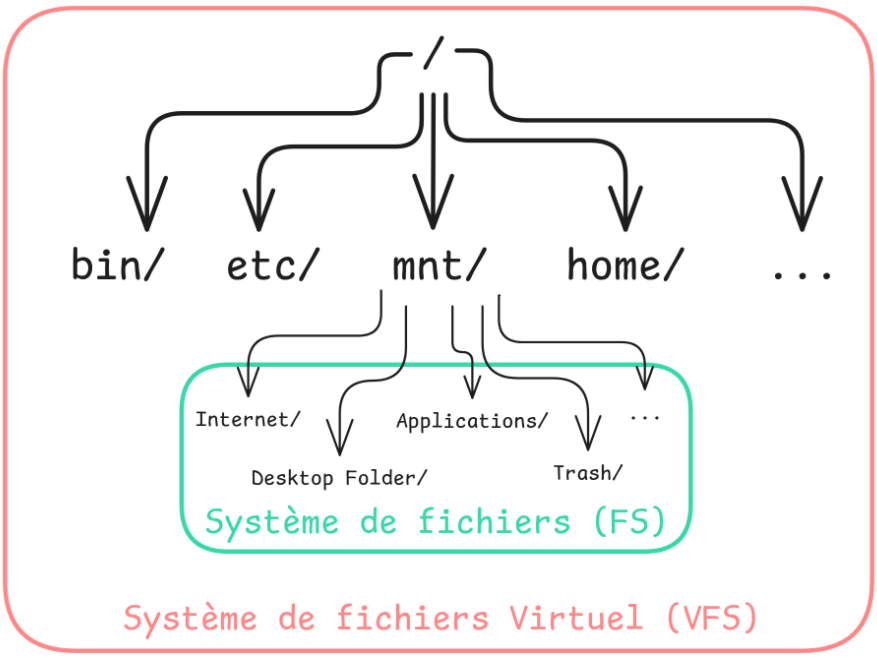

FIGURE 5 – Schéma représentant le système de fichiers virtuel (*VFS*), et un système de fichier concret comme branche du *VFS*.

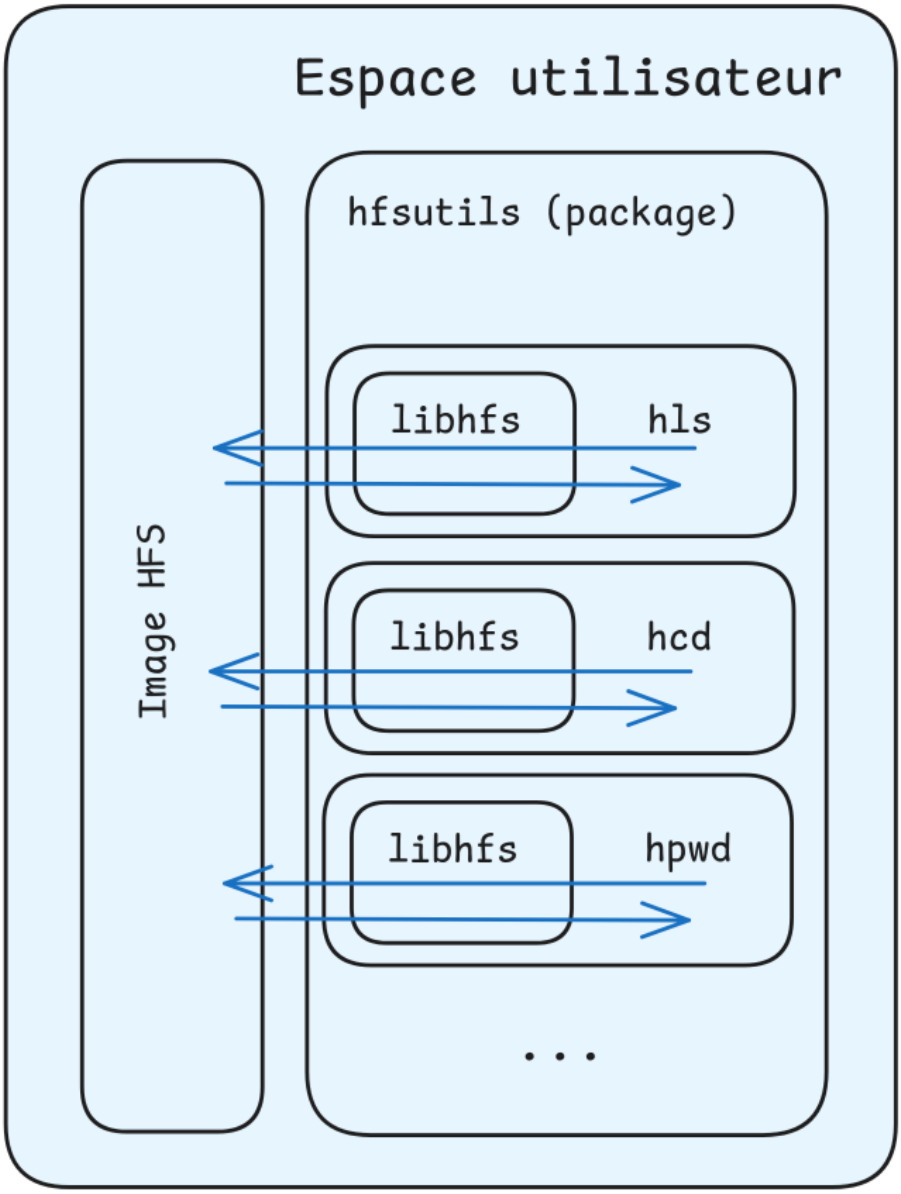

FIGURE 6 – Fonctionnement des programmes de hfsutils.

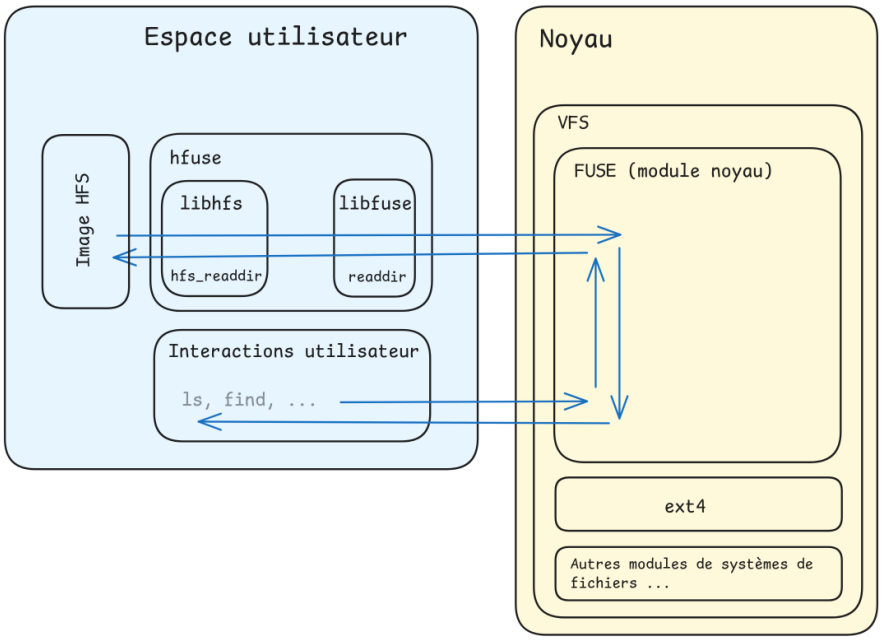

FIGURE 7 – Schéma de communication pour *hfuse*.