Ecole Publique d'Ingénieurs en 3 ans

Rapport fin de parcours

PLATEFORME FORENSIQUE

Le 18 Février 2021,
Version 1.0

Arthur ROUILLÉ, Bastien HUBERT,
Nicolas VIRARD et Léo MÉTAIS

Tuteurs : Christophe ROSENBERGER,
Lyes KHOUKHI

www.ensicaen.fr

TABLE DES MATIERES

TABLE DES FIGURES

PRESENTATION DU PROJET

1. Contexte

La forensique regroupe l'ensemble des différentes méthodes d'analyse fondées sur les sciences afin de servir au travail d'investigation de manière large. Nous nous focalisons dans ce document plus particulièrement sur les traces numériques laissées par un utilisateur lors de l'usage d'un appareil connecté ou d'un logiciel (incluant les navigateurs Web). Le terme trace numérique est utilisé dans les domaines de la sécurité, de l'informatique légale et des systèmes d'information. Il désigne les informations qu'un dispositif numérique enregistre sur l'activité ou l'identité de ses utilisateurs au moyen de traceurs tels que les cookies, soit automatiquement, soit par le biais d'un dépôt intentionnel : moteurs de recherche, blogs, sites de réseau social, sites de e-commerce, mais aussi cartes à puce, titres de transport, téléphones mobiles.

Tous les systèmes qui requièrent une identification ou une interaction sont susceptibles de capter des informations sur l'utilisateur – parcours, requêtes, préférences, achats, connexions, évaluations, coordonnées. Les traces ne sont pas des messages, mais des données (typiquement des fichiers de log). Prises isolément, elles n'ont guère de sens, mais regroupées, traitées et combinées dans d'importantes bases de données, elles peuvent révéler des informations significatives, stratégiques ou sensibles. La notion de traçabilité numérique est de plus en plus présente dans nos sociétés, cela est dû au contexte actuel des Big data, toute information (data) est enregistrée et stockée par défaut.

Les traces numériques peuvent en particulier être utilisées pour profiler les personnes, par extraction automatique d'un profil à partir de l'observation de leurs comportements. Ce profilage peut servir ensuite à faire du ciblage comportemental, très utile au marketing sur le web. L'analyse de ces traces numériques peut se révéler être très utile pour une investigation soit criminelle (récupération d'éléments de preuves dans une affaire criminelle) soit en cas d'incident (identification de la raison d'une défaillance liée par exemple à un virus). Une dernière application concerne la mesure d'attente à la vie privée d'un utilisateur (dans quelle mesure ces traces permettent d'identifier un individu sur internet par exemple).

2. Objectifs d'une plateforme forensique

Le développement d'une plateforme forensique présente deux avantages majeurs. D'une part, une plateforme logicielle en forensique pourrait permettre de réaliser des analyses de haut niveau et automatisable pour des opérationnels de l'investigation (section recherche criminalité numérique de Caen par exemple) et pour des chercheurs (par exemple pour des activités de recherche en protection de la vie privée).

D'autre part, une telle plateforme serait un outil intéressant comme support de projets pour des étudiants en informatique pour les faire travailler sur des sujets en lien avec la sécurité, les systèmes d'exploitation, le réseau… Elle pourrait être utilisée à l'ENSICAEN ou dans le cadre du Master E-Secure de l'UNICAEN.

Aussi, les compétences du GREYC en traitement automatique de la langue, traitement des images, sécurité, etc. doivent permettre de développer de nouveaux outils de haut niveau pour des investigations numériques.

3. Déroulement du projet

Ce projet de grande ampleur, regroupe de nombreuses personnes de tous horizons. En plus de l'équipe de troisième année que nous formons - *Léo Métais, Nicolas Virard, Bastien Hubert et Arthur Rouillé* - des étudiants de première année, de master E-Secure mais aussi des professionnels de l'investigation (*Gendarmerie Nationale*, experts judiciaires), des enseignants et des chercheurs donnent cœur à l'ouvrage.

Toutes les deux semaines depuis le lancement du projet sont organisées des réunions, avec un tour de table et des discussions pour rétablir les objectifs de chacun. Ainsi, le projet se veut autonome, et toujours plus consolidé par une approche quasiment "Agile". En ce qui nous concerne, étudiants de troisième année, nous pouvons prévoir l'organisation du projet comme suit :

Prénom	**Tâches**
Arthur	Filtre de nus Serveur Python
Bastien	Interface graphique Parallélisation
Nicolas	Gestionnaire de filtres Base de données Architecture logicielle
Léo	Filtre de duplicatas Serveur Python

Afin d'identifier le travail fourni par l'ensemble des collaborateurs sur le projet, pour la plupart de l'ENSICAEN, nous avons trouvé cela intéressant de jouer sur la coïncidence entre le

mot « forensique » et le nom de l'école. Ainsi, la suite de ce présent rapport fera mention à la « plateforme forENSIque ».

LISTAGE DES FICHIERS ET INTERFACE GRAPHIQUE

1. Listage des fichiers

1.1. Une tâche basique mais indispensable

La nécessité de lister tous les fichiers d'un disque rapidement est évidente en ce qui concerne une analyse forensique. En effet, dans le cadre d'investigations judiciaires, le temps est souvent un facteur déterminant dans la réussite de l'enquête. Dès lors, il est primordial d'être en mesure de scanner intégralement un dispositif de stockage, ne serait-ce que pour avoir une idée très générale de la structure des données ainsi que leur volume. Bien entendu, la rapidité de l'opération varie grandement selon le type de technologie de stockage et l'on est limité par le matériel lui-même. Un HDD sera bien plus long à analyser qu'un SSD. Raison de plus pour effectuer une analyse préalable de la structure du disque afin de pouvoir travailler dessus. En outre, nous avons choisi de considérer notre arbre de fichier comme un arbre de type Linux, où fichiers et dossiers sont indifféremment traités en tant que fichiers.

1.2. Première itération du parcours de fichiers

La première version de notre programme utilise un algorithme séquentiel de parcours de disque. Ainsi, un seul "thread" du programme entre dans le premier dossier, cherche s'il y a des sous-dossiers, puis les explore en priorité : le parcours se fait en profondeur d'abord.

1.3. Seconde itération du parcours de fichiers

Dans une 2e version de notre programme nous avons décidé d'utiliser le multithreading afin de pouvoir améliorer les performances. Pour cela nous avons employé une méthode similaire mais répartissant la charge de travail sur plusieurs cœurs (physiques ou logiques) de la machine. En effet, au lieu d'explorer un seul dossier récursivement, les différents cœurs s'attribuent respectivement des dossiers. On utilise ainsi un "cœur" par dossier à explorer dans le meilleur des cas. Cela est ainsi d'autant plus efficace que les dossiers possèdent beaucoup de contenu.

1.4. Conclusion

On s'aperçoit que le gain de vitesse est significatif avec un SSD, bien moins avec un HDD. Ces derniers ont l'inconvénient d'être grandement limités par leur vitesse de transfert. Le multithreading permet pour un SSD d'atteindre aussi la limite de transfert de ces derniers.

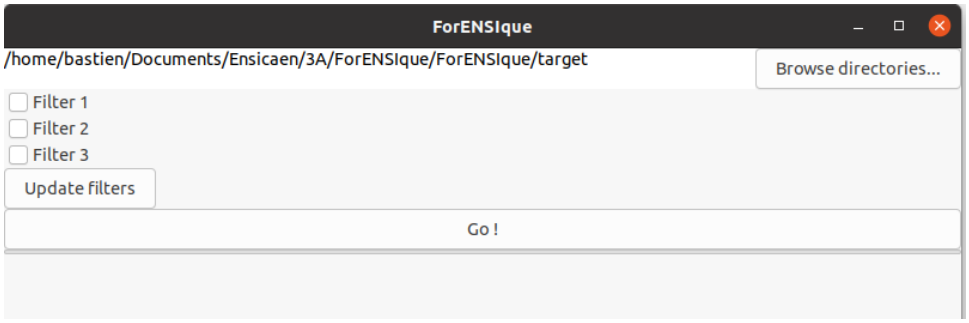
2. Conception de l'interface graphique

2.1. Choix du kit de développement de l'interface graphique

La première étape était de déterminer un kit de développement suffisamment simple d'utilisation mais aussi bien documenté pour pouvoir créer aisément constituer une interface graphique. Nous sommes bien entendu restés sur du Rust, et avions le choix entre Qt et GTK. Après avoir essayé non sans difficulté le premier, nous avons finalement porté notre choix sur GTK.

2.2. Création de la partie graphique

L'interface est très basique. Nos successeurs auront certainement à utiliser abondamment des menus afin de mieux ranger les options qui s'offrent à l'utilisateur. Nous avons utilisé le logiciel *Glade* afin de générer simplement un fichier XML aisément modifiable manuellement si besoin.



2.3. Intégration de la partie métier dans l'interface graphique

Cette étape, la plus délicate, est loin d'être finie car dépendant fortement des avancées constantes sur le code "métier". La première tâche à réaliser est de pouvoir faire sélectionner à l'utilisateur un dossier, puis lui permettre de d'appliquer les traitements qu'il souhaite à ceux-ci. Pour ce faire, il est généralement nécessaire de passer par un patron d'architecture de type MVP (Modèle-Vue-Présentation). Dans notre cas, le Modèle est notre partie métier, développée en amont. La Vue doit être un ensemble de fonctions faisant le lien entre les deux autres parties : c'est un adaptateur. Enfin, la Présentation est la partie purement graphique de notre application, ne contenant peu ou pas de logique, et regroupe un fichier XML nécessaire à la création de l'interface, ainsi que quelques fragments de codes pour instancier des objets lui étant propres.

Le choix du langage Rust comporte certains mécanismes lourds (les emprunts par exemple) qui rendent l'écriture du code de la Vue peu aisé, et est potentiellement source de bogues. D'un autre côté, on gagne en performance du fait que le langage est compilé et non interprété. Cela est d'autant plus vrai que l'on peut compiler une version de production de l'applicatif, bien mieux optimisée.

2.4. Conclusion

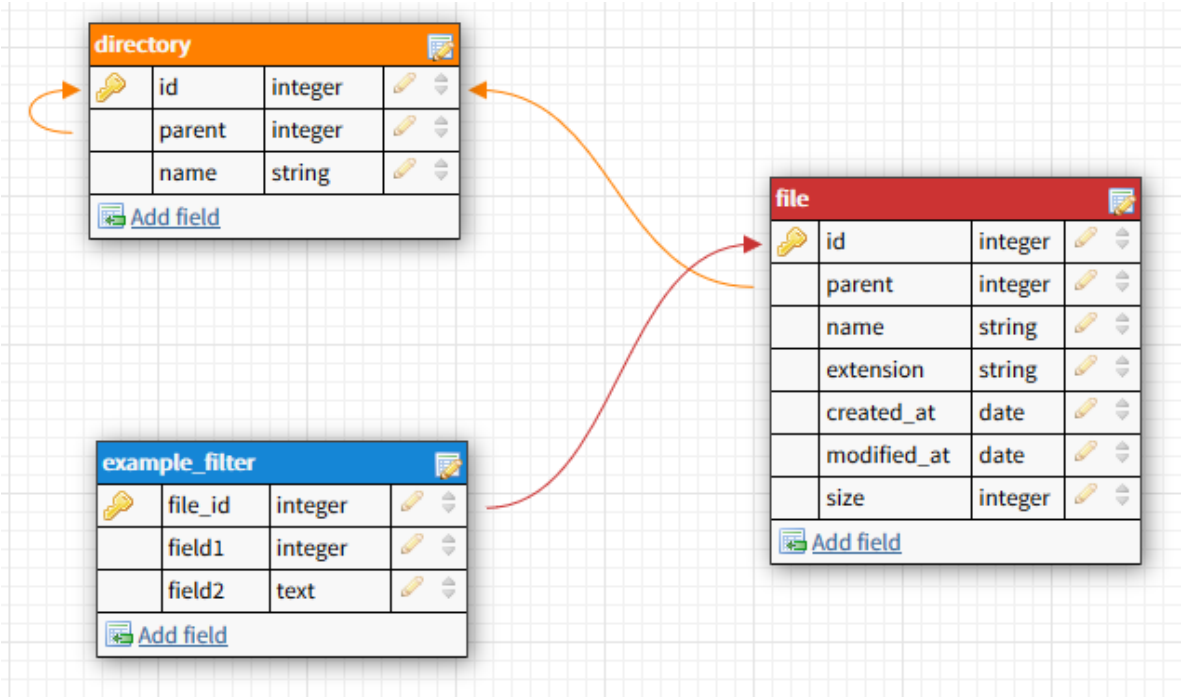
Encore beaucoup de choses doivent être faites pour parvenir à une interface graphique réellement efficace et reflétant bien l'avancée de notre application. Néanmoins, le but premier n'est pas d'obtenir à tout prix une interface graphique, mais bien plus de poser des bases solides pour son expansion future. C'est dans cette perspective que nous développerons en priorité des fonctions de visualisation de l'avancement des tâches, avant d'implémenter progressivement toutes les fonctionnalités abouties de la ligne de commande à l'interface utilisateur. En outre, il peut tout aussi bien être possible de développer une interface graphique web, peut-être plus simple à mettre en place qu'une interface graphique entièrement compilée.

MISE EN PLACE D'UNE BASE DE DONNEES

1. Contexte

La gestion des données est une partie importante du projet. Il faut avoir une structure optimisée afin de pouvoir faire des recherches diverses sur les données récoltées. Ainsi, une base de données est naturellement une solution adaptée au problème. Une base de données permet en effet de stocker les données de façon optimisée pour des requêtes futures : il est alors possible de faire n'importe quelle requête pour récupérer des données utiles.

2. Schéma de la base de données



3. Conception

Plusieurs choix ont été adopté afin de concevoir la base de données. Il y a d'abord une première table "directory" qui correspond à un dossier. Celui-ci à un id, un nom et un parent. Ce parent correspond à un dossier lui-même, car un dossier peut être inclus dans un autre dossier naturellement. Ensuite, une table "file" a été créée : elle correspond à un fichier qui possède un id, un nom, une extension, une date de création, une date de modification, une taille et un parent, qui est un dossier. Notez que plusieurs choix ont été fait : l'extension a été séparée du nom pour pouvoir rechercher rapidement les fichiers d'une certaine extension et les fichiers et dossiers sont hiérarchisés en fonction de l'id du parent et non d'un chemin

textuel. Ceci permet plusieurs choses, la première est qu'il est facile de trouver les dossiers et fichiers dans un même dossier, grâce à l'id, car il n'est ainsi pas nécessaire de découper une chaîne de caractère pour chaque champ pour chaque requête, ce qui est une opération très coûteuse. Deuxièmement, la hiérarchisation n'est plus dépendante de toute la chaîne mais juste d'un id du parent, il est ainsi facile de déplacer des fichiers ou dossiers vers un autre dossier. Enfin le chemin est relatif à la racine (id : 0) et il est relativement facile de le reconstituer récursivement le chemin textuel pour un affichage pour un humain. Enfin, on constate une table "*example_filter*" qui contient une référence vers un fichier ainsi que des champs utiles. En pratique, une multitude de tables peuvent être créées en fonction des filtres utilisés.

4. Résultats

Les résultats pratiques montrent qu'une base de données est en effet une solution adaptée à la gestion des données du programme par sa structure permettant des requêtes et sa persistance sur le disque. À l'heure où ce rapport est écrit, seules les base de données sqlite3 sont supportées. Cependant, grâce à une structure du programme modulable, il est facile d'ajouter d'autres types de base de données : MySQL, PostgreSQL, etc... Ces autres types de bases seront très certainement ajoutés dans les prochaines versions du programme.

CREATION DE FILTRES SUR LES DONNEES

1. Pour un traitement efficace des données

1.1. Objectifs

Durant ou à la suite du listage des fichiers, nous souhaitions ajouter des filtres qui trieraient ou signalerai les fichiers d'intérêt. Chaque filtre ayant un but différent ainsi qu'un fonctionnement qui lui est propre, nous devions concevoir la plateforme de manière modulaire pour que l'ajout ou le retrait d'un filtre soit le plus simple possible.

La possibilité d'appeler des programmes externes, des scripts de n'importe quel langage, ou simplement une extension de notre propre création était le défi principal de la gestion des filtres.

1.2. Etat de l'art

Beaucoup des filtres que nous avons imaginé pour la plateforme existe déjà sous différentes formes, algorithme et langage de programmation. Pour certains, il est simplement possible d'appeler un programme existant, pour le reste, il reste intéressant de connaître les solutions existantes.

Le framework *Autopsy* étant une solution open source de référence dans la forensique digitale, nous avons pu l'étudier pour savoir quel filtre et quelles fonctionnalités étaient importante pour notre plateforme. Nous avons pu en retirer de sérieuses pistes pour les filtres d'extension incorrectes et de fichiers chiffrés.

Pour les filtres suivants, nous faisions des recherches pour chacun. *GitHub* était une source très utile pour faire nos recherches sur l'état de l'art pour chaque fonctionnalité de notre plateforme. Les filtres de quasi-doublons et de nus s'appuient sur des solutions open-source étudiés via *GitHub*. Une adaptation conséquente était nécessaire pour chacun de ces filtres afin de garantir leur fonctionnement et un temps d'exécution minimal.

2. Développement d'un gestionnaire de filtres

2.1. Contexte

Le but de ce projet est de réaliser un programme supportant autant de filtres qu'il est possible d'en créer. Ainsi, il va se soit qu'il faille un programme modulable, auquel il est facile de greffer des nouveaux filtres et de pouvoir exécuter seulement les filtres voulus. C'est pourquoi il a été développé au sein du programme un gestionnaire de filtres, qui s'adapte à n'importe quel nouveau filtre.

2.2. Conception

Le gestionnaire de filtre a été développé afin de pouvoir traiter chaque filtre de la même manière. Ainsi il remplit les rôles suivants :

- Enregistrer les différents filtres
- Activer et désactiver les filtres en fonction des paramètres du programme
- Envoyer les requêtes de création des tables des filtres
- Envoyer les fichiers à traiter dans chaque filtre et insérer les résultats dans la base de données
- Calculer les performances des filtres (en nanosecondes)
- Gérer les serveurs locaux pour utiliser d'autres langages (ex: Python)

Le gestionnaire enregistre donc d'abord les structures des filtres. Ensuite, les paramètres du programme sont passés au gestionnaire de filtre afin que celui-ci active les filtres demandés : cette demande dépend de ce que l'utilisateur souhaite utiliser ou non. Ensuite, le programme demande les tables à créer au gestionnaire de filtre qui sont ensuite passées à la base de données afin d'être créées. Ensuite vient la boucle principale du programme, chaque fichier itéré est envoyé en traitement au gestionnaire de filtre qui se charge de faire passer ce fichier dans chaque filtre et d'insérer les résultats obtenus dans la base de données. Chaque filtre a une phase de traitement, qui calcul le résultat, et de post-traitement, qui permet d'analyser ce résultat et d'insérer dans la base ce qu'il faut. Ces deux phases sont mesurées en temps pour des soucis de calculs de performances, uniquement si l'utilisateur précise qu'il souhaite avoir les performances.

Dans le cas de certains filtres, par exemple le filtre de nus, nous avons eu le besoin d'implémenter des filtres dans d'autres langages que le Rust, ici le Python. Pour cela nous avons implémenté des serveurs locaux modulables.

2.3. Serveurs locaux

Lors de l'activation des filtres indiqués, certains peuvent être implémentés non pas en Rust mais dans d'autres langages. Pour répondre à cette problématique, nous avons mis en place un gestionnaire de serveurs locaux. Un serveur gère tous les filtres correspondant à un langage (par exemple, un seul serveur gère tous les filtres Python, un autre tous les filtres d'un autre langage etc.). Afin d'avoir un programme modulable et qui puisse communiquer de la même façon avec tous ces serveurs, il a fallu implémenter une communication normalisée.

Au démarrage du serveur, le port à utiliser est passé en argument du programme. Ainsi, le serveur se doit de démarrer une écoute TCP sur le port en question et une autre sur le port donné + 1. Un canal permet d'avoir une communication Rust -> Serveur et un autre une

communication Serveur -> Rust. Ceci permet de ne pas être bloqué par l'écoute ou par l'envoi de données (étant donné que tout se fait en multithreading).

Ensuite, tout se passe sur l'échange de paquets avec des formats spécifiques. Un paquet reçu par le serveur aura le format suivant :

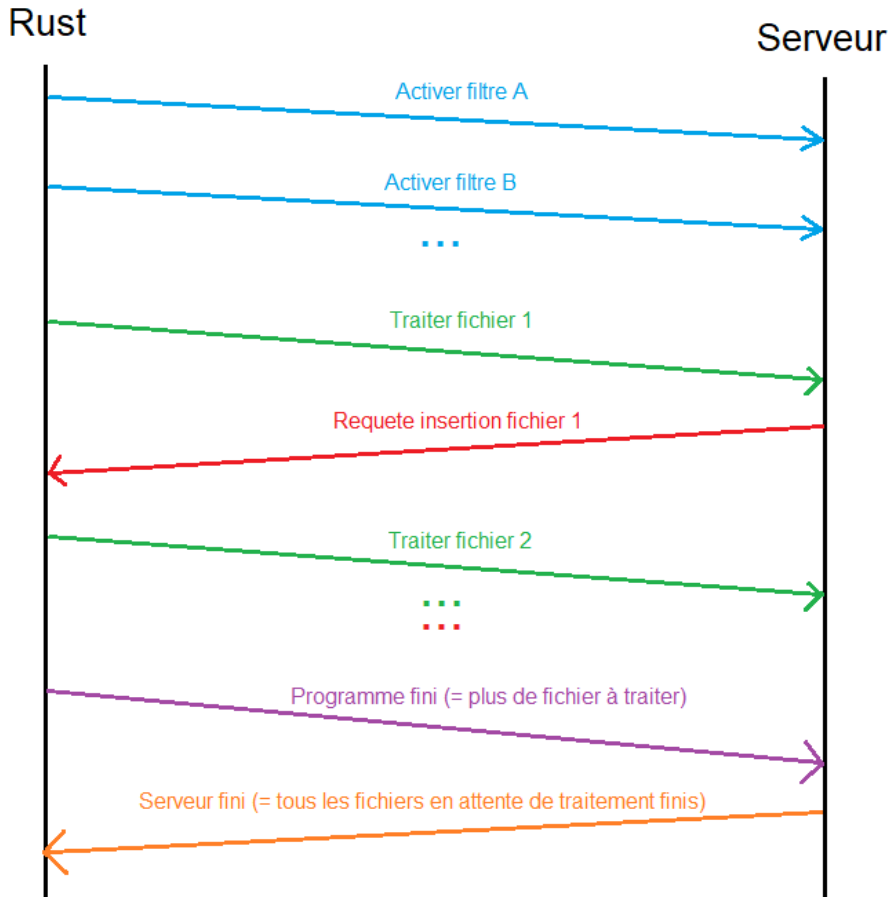
Taille de l'id + données (2 octets)	Id du paquet (1 octet)	Données (longueur variable)

Dans le cas inverse (reçu par le serveur Rust), la taille ne doit pas être renseignée. Cela a été fait de façon à ce que le serveur puisse découper les requêtes, car il fonctionne par stream. Cela n'a pas été implémenté du côté du Rust car le stream est très rapide et ne sature jamais.

Les ids et les données sont les suivants :

Nom	**Direction**	**ID**	**Données**
Activer un filtre	C -> S	0	Nom du filtre (utf8)
Traiter un fichier	C -> S	1	Id du fichier (i64), Chemin du fichier (utf8)
Fichier traité	S -> C	2	Requête SQL à exécuter
Programme Rust fini	C -> S	3	/
Serveur fini	S -> C	4	/

Ainsi, la communication se passe ainsi :



2.4. Résultats

Cette structure de gestionnaire de filtre permet de répondre au besoin de pouvoir ajouter des filtres de façon modulable et indépendamment des langages utilisés sur les serveurs. Cependant il reste quelques améliorations possibles : calculer les performances sur les serveurs distants et ne plus utiliser le langage SQL qui limite d'autres base de données avec d'autres langages pour le moment.

3. Développement de filtres

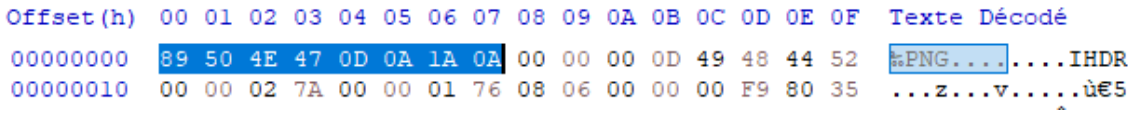
3.1. Détection de types de fichiers

3.1.1. Contexte

L'un des premiers filtres qui a été jugé nécessaire au programme est celui de détection de type de fichiers et d'extension. En effet, il est facile pour un utilisateur de changer l'extension d'un fichier en un autre, afin de tromper les investigations, par exemple changer une extension .jpg en .docx afin de ne pas pouvoir ouvrir ce fichier et ainsi visualiser son contenu qui peut être incriminant.

Cependant, il faut savoir qu'il est facile de détecter certains types de fichiers grâce à ce qu'on appelle les nombres magiques. Effectivement, les formats de fichiers courants possèdent des

octets qui peuvent permettre de les identifier. Par exemple une image jpg commencera toujours par les octets "89 50 4E 47 0D 0A 1A 0A", un média .avi (Audio Video Interleaved) par "52 49 46 46 ?? ?? ?? ?? 41 56 49 20" etc... Le but de ce filtre est donc de trouver des nombres magiques afin de déterminer le format réel d'un fichier, peu importe son extension. Dans l'image ci-dessous, nous reconnaissons les octets d'un fichier png, nous pouvons donc en déduire son format réel.



3.1.2. Mise en place

Plusieurs techniques d'optimisation ont été mise en place afin d'avoir un filtre rapide et efficace. Les nombres magiques ont une taille de 2, 3, 4, 6, 8, 12 ou 16. Le filtre ne lit donc que les 16 premiers octets d'un fichier. Ensuite une *hashmap* est utilisée pour trouver le format en temps quasi constant (O(1)). Quelques formats ont nécessité des ajustements, où le nombre magique variait, comme les RIFF (*avi*, *wav*...) ou les FORM (*aif*, *yuv*...) ou encore les format "tar" où le nombre magique se trouve à l'octet n°0x101. Ainsi, le filtre ne lit que le minimum nécessaire du fichier et trouve le format en temps quasi constant. Ceci permet d'avoir un filtre très rapide comme expliqué dans les résultats ci-dessous.

3.1.3. Résultats

L'implémentation de ce filtre permet ainsi d'avoir des performances très élevées. On constate en moyenne un temps de traitement de 45000 ns et un temps d'insertion de 24000 ns. Ceci permet de traiter en théorie 15000 fichiers par seconde ce qui est un taux très élevé. Cependant, il ne faut pas oublier que le reste du programme et des filtres peuvent prendre des cycles CPU ce qui réduit ce taux en pratique.

Ce filtre permet ainsi de détecter le format réel d'un fichier et de le comparer à son extension. Si un fichier possède un format détecté différent de l'extension, un flag sera actif dans la base de données pour signaler que ce fichier est suspect. Le filtre supporte 134 formats de fichier actuellement, des plus au moins courants.

3.2. Détection de nus

3.2.1. Contexte

Durant les réunions où des experts en investigation criminelle étaient présents le besoin de détecter rapidement des photos suspecte a été mis en exergue. Alors, sous avons convenu de mettre en place un filtre permettant de relever les images pouvant être des photos de personnes dénudées ou de caractère érotique. De plus, il est apparemment de plus en plus fréquent d'avoir des images de synthèse sur ces thématiques.

Afin de détecter ces images, que nous allons généralement dénommer « images de nus », nous allons nous appuyer sur une technologie qui fait ses preuves sur les réseaux sociaux. Les algorithmes d'apprentissage profond permettent de détecter rapidement

Finalement, la solution retenue est la librairie *Deep NN for NSFW Detection*[1], publiée sur GitHub. D'après la documentation de l'auteur, nous avons ici un réseau de neurones artificiels ayant été entrainé sur plus de 60 Go de données. Il est aussi capable de catégoriser les images sous les catégories suivantes :

- **Neutre :** photographies sans prétention de choquer
- **Dessins :** dessins sans prétention de choquer
- **Hentai :** dessins à connotation pornographique
- **Pornographie :** images pornographiques, actes sexuels
- **Images sexuellement explicites,** pas de pornographie

Par ailleurs, le script permet pour une image d'un chemin donné d'obtenir une réponse en syntaxe *JSON* de la forme d'un doublet (chemin, scores) où les scores sont les catégories énoncées ci-dessous ayant une valeur comprise entre 0 et 1. Ainsi, une image où le score *Neutre* est la plus élevée que les autres, alors c'est probablement une image qui n'est pas suspecte.

3.2.2. Mise en place

La librairie utilisée a été développée en Python. Alors, il nous a fallu dans un premier temps mettre en place une adaptation afin que notre plateforme puisse communiquer aisément avec le détecteur de nus. Pour ce faire, nous avons mis en place une structure basée sur l'exploitation de fichiers de travail, contenant du JSON. En effet, ce format de données est universel et est compris nativement par Python et Rust. Le schéma ci-dessous décrit le fonctionnement du filtre. Notons que l'étape 4 est répétée pour chaque image.

[1] https://github.com/GantMan/nsfw_model

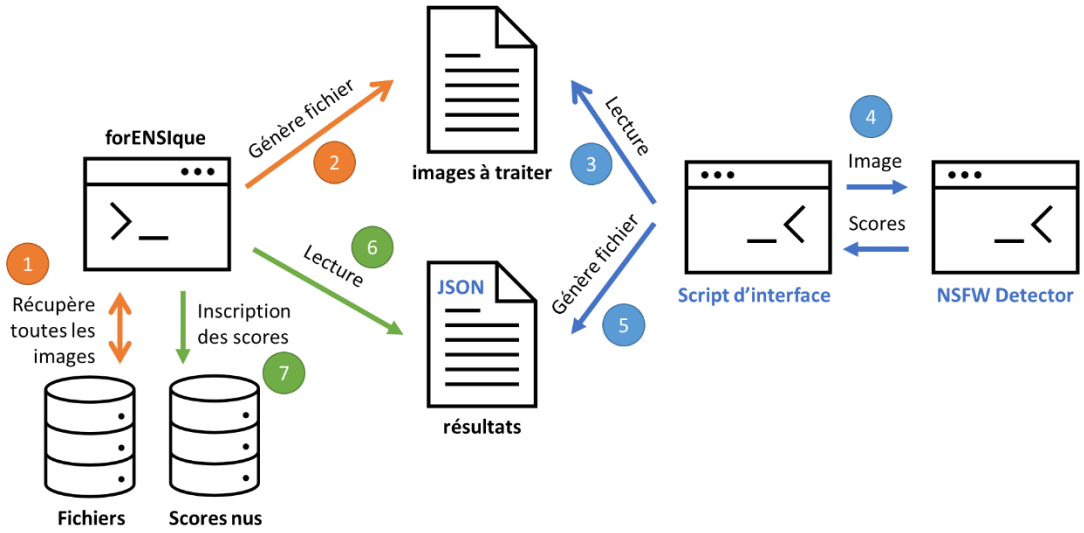


Figure 1 - Communication plateforme / script pour le filtre de nus

Cette solution fonctionne correctement, mais contrairement aux autres filtres qui sont appelés fichier par fichier, celui-ci est lancé a posteriori. Comme nous allons le voir par la suite, l'étape 2 génère un fichier contenant l'ensemble des chemins d'image à traiter. Pour ce faire, nous avons mis en place une requête SQL récursive qui reconstruit le chemin d'un fichier de dossier parent en dossier parent.

3.2.3. Résultats

À la suite de l'implémentation, nous avons pu faire un essai sur des images trouvées sur Google Image. Pour ce test le script passe 3.2 secondes pour 17 images soit 188 ms par image. Qualifions à présent les performances de détection :

- Pour l'échantillon photos "normales" : 5/5 neutre
- Pour l'échantillon "douteuses" : 2/2 neutre
- Pour l'échantillon photos "nu" :
    - 6/10 nu,
    - 2/10 neutre (2 photos d'homme)
    - 1/10 dessin (devrait être une photo de nu)

Maintenant, qualifions les performances de calcul, sur répertoire utilisateur (/home/user) qui contient des images de différentes sources (fond d'écran, photos du téléphone portable mais aussi des miniatures et des images provenant du cache). Ainsi, nous avons :

- Pour la détection (module Python) :
    - 120 secondes pour 1460 images

- 82 ms par image

- Pour la reconstruction des chemins de fichiers :
  - 430 secondes (7mn20s)
  - 300ms par fichier

Autrement dit, nous avons 72% du temps du traitement consacré à la reconstitution du chemin d'un fichier à partir de la base de données ce qui est bien trop. Sachant qu'une détection via un réseau de neurones est censé être plus lent qu'une requête SQL.

3.2.4. Nécessité de la mise en place d'un serveur local

Afin d'être cohérent architecturalement, il faudrait alors adapter le fonctionnement du filtre, tel que décrit dans le chapitre « création de filtres sur les données ». La solution consiste donc à s'épargner des contraintes techniques du script (ici du Python), en se contentant seulement d'envoyer une demande de type « quels sont les scores de nus de l'image ayant le chemin *xxxxxx* ? ». Il s'agit alors des propos évoqués dans la partie « 2.3. Serveurs locaux ».

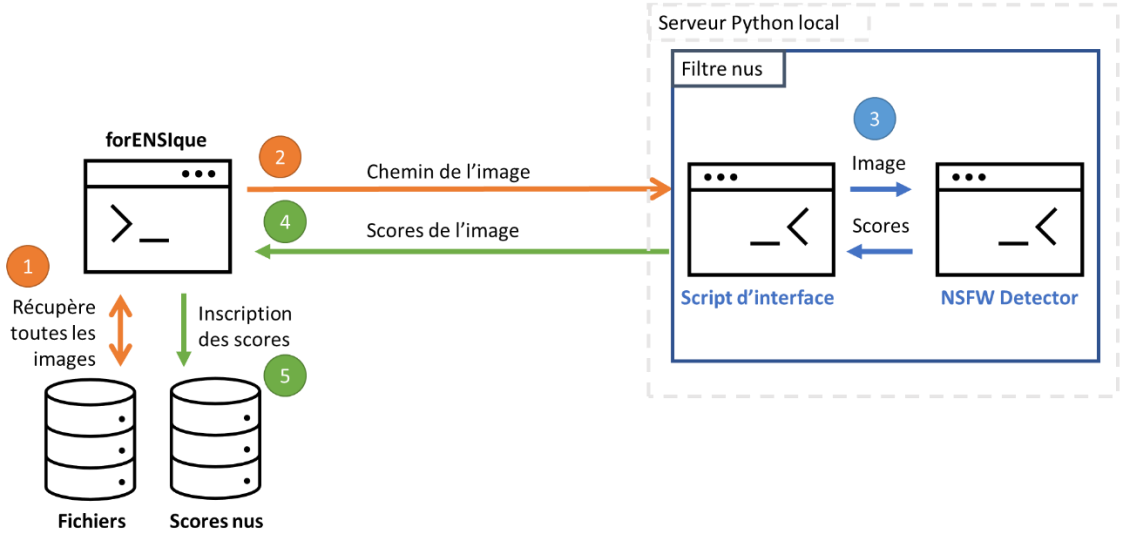


Figure 2 - Fonctionnement du filtre de nus obtenu

Après l'amélioration de la requête SQL permettant de reconstruire le chemin d'un fichier, nous obtenons un gain de performances considérables, alors qu'un chemin se reconstruit en moins d'une milliseconde en moyenne, la classification d'une image prend moins de 100ms en moyenne.

3.3. Récupération de données de navigation

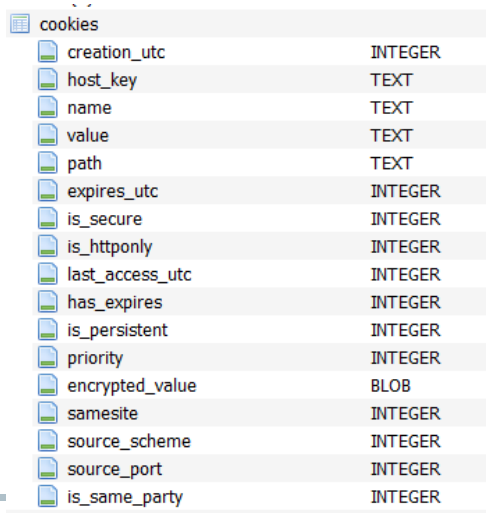
Ce filtre a pour but de récupérer les bases de données générées par les navigateurs web afin d'obtenir les cookies de connexions et l'historique de navigation de l'utilisateur.

3.3.1. Bases de données

Les navigateur web génèrent deux bases de données auxquelles nous nous sommes particulièrement intéressés : la base des cookies et la base de l'historique. Les navigateurs gèrent chacun différemment leur base de données mais les mêmes informations s'y retrouvent. Le programme est capable de détecter les navigateurs de type Firefox et Chromium (Chrome, Edge, Brave, …). Le navigateur Safari n'est pas encore pris en compte. Les exemples sont donnés pour le navigateur Brave.

3.3.1.1. La base des cookies

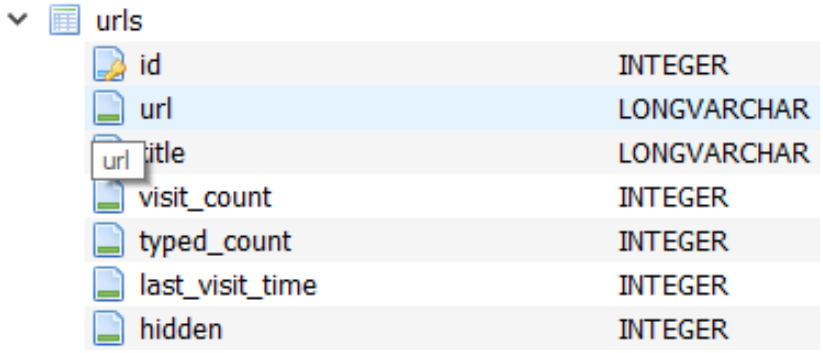
La base de données stocke les cookies de la manière suivante :



Pour l'instant, seuls le site web (host_key), la date d'expiration du cookie (expires_utc), la date de dernier accès au cookie (last_access_utc) et la date de création du cookie (creation_utc) sont récupérer par le projet. Les autres informations ne sont pas traitées car ne sont soit pas communes aux différents navigateurs, soit non pertinentes pour l'utilisation actuelle.

3.3.1.2. La base de l'historique

La base de données stocke l'historique de la manière suivante :



De la même manière, seuls l'url, le nombre de visite de cet url (visit_count), et la date de dernière visite (last_visit_time) sont enregistrés.

3.3.2. Intégration du filtre

La forme de filtre pour cette fonctionnalité n'est pas évidente. Pour intégrer la récupération des données de navigation, le filtre vérifie si le fichier analysé dans la boucle principale est une BDD exploitable. Pour ce faire le filtre vérifie

le chemin du fichier et le compare à un modèle regex prédéfini qui reconnait les BDD connus et trouve dans un second temps le navigateur associé.

Par exemple « `.config/BraveSoftware/Brave-Browser/Default/Cookie` » sera reconnu comme une base de données exploitable, puis la chaine de caractère « `BraveSoftware/Brave-Browser` » déterminera que la base a été généré par le navigateur Brave qui est un Chromium.

Ensuite, les données récupérées sont gérées par le gestionnaire de filtres qui se charge de créer une nouvelle table et d'ajouter les données extraites à la base de données. Ainsi les informations rassemblées par ce filtre rejoint les informations extraites par les autres filtres.

3.3.3. Amélioration

Le navigateur Safari possède une importante part du marché des navigateurs web, il sera donc important de le prendre en compte et de rajouter les bases de données de ce navigateur. D'autres navigateurs moins importants pourront également être rattaché. Il est tout de même notable que tous les navigateurs issus du noyau libre de droit Chromium sont déjà pris en compte car ont une architecture commune.

Il est possible de récupérer un certain nombre de données dans les bases de données, et tous les champs n'ont pas été exploité. On pourra imaginer des usages plus spécifiques qui feront usage d'autres de ces informations.

Enfin, outre les données non utilisées, il existe d'autres tables et d'autres bases de données non-exploité qui peuvent être intéressante pour trouver toujours

3.4. Détection de duplicatas

Ce filtre s'appuie sur une technique de hash sensible au modifications locales[2] et plus précisément sur le *dHash*. Plusieurs fonctions de hachage du même type existent, notamment le *pHash* (étudié mais non retenu à cause du temps de calcul) et le *aHash* (bien trop lourd également).

3.4.1. Mise en place

Ce filtre, comme d'autres, utilise principalement le langage de programmation python. Pour des raisons d'efficacité, nous avons mis en place un serveur local python qui tourne dans un thread à part, dans l'attente de code python à exécuter. Dans ce cas, pour minimiser le temps de calcul du programme, on donne en entrée un dossier entier d'images.

[2] "Near-duplicate image detection using Locality Sensitive Hashing": https://realpython.com/fingerprinting-images-for-near-duplicate-detection/

Ces images voient leurs *dHash* calculés puis comparés les uns aux autres. Une fois ce processus terminé, le programme va chercher des paires d'images au hash semblables. Une distance de Hamming est ici mise en jeu pour mesurer la distance entre deux hash.

Une valeur prédéfinie sert de seuil pour définir si les hash des images sont suffisamment proches pour être définie comme quasi-identiques. Le serveur python renvoie ensuite les quasi-doublons pour qu'ils soient labélisés en tant que tels dans le programme Rust.

3.4.2. Améliorations pour la suite du projet

Pour la suite du projet, pour des soucis de rapidité d'exécution, il faudra minimiser la quantité de données transitant entre le programme principal et le serveur python. Cette amélioration sera une priorité.

Il pourra également être intéressant de changer le fonctionnement et tenter de calculer les hash de images une à une plutôt de de faire tout un dossier en un coup. Le résultat ne sera pas forcément plus efficace, mais il est important de chercher des solutions différentes pour être sûr de minimiser le temps d'exécution.

Enfin ce filtre n'est pas encore intégré, ni au gestionnaire de filtre, ni à l'architecture en serveur. Il pourra donc être possible de rattacher ce programme au reste de l'architecture.

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Ecole Publique d'Ingénieurs en 3 ans

6 boulevard Maréchal Juin, CS 45053
14050 CAEN cedex 04

ministère éducation nationale enseignement supérieur recherche

Normandie Université

Cti Commission des titres d'ingénieur

cdefi Conférence des Directeurs des Écoles Françaises d'Ingénieurs

European Accreditation of Engineering Programmes EUR-ACE®

CONFÉRENCE DES GRANDES ÉCOLES