Projet 2A : Ext4 en Go

Yahya Chikar, Antony Huynh, Abdulaziz Kalash, Yam Pakzad, Maelys Sable

23 mars 2025

Contents

1 Présentation générale du projet et du contexte

Notre projet consiste à implémenter en Go une librairie permettant d'analyser une partition d'un disque dur formatée en Ext4, HFS+ et/ou APFS.

1.1 Présentation du contexte

1.1.1 Le langage Go

Go est un langage développé par Google, dont la première version date de 2009. Il est utilisé dans de nombreux projets d'applications open source comme Docker ou Kubernetes, pour sa simplicité de syntaxe et ses performances élevées notamment grâce au multithreading. La librairie standard du Go est très large et couvre une grande partie des besoins d'un projet, de la programmation bas niveau jusqu'à la mise en œuvre d'une API. Par ailleurs c'est le langage principalement utilisé par la Gendarmerie Nationale pour développer ses logiciels et son backend.

Utiliser le Go pour écrire cette librairie était donc assez pertinent, au vu des possibles futures utilisations de la librairie

1.1.2 Les utilisations de la librairie

En effet, l'analyse d'une partition d'un disque peut donner énormément d'informations techniques, comme les contenus des fichiers, leurs métadonnées, leur offset dans la partition, les clusters occupés et libres de la partition. Une grande partie de ces informations peut être exploitée par les forces de l'ordre comme la Gendarmerie Nationale pour des analyses forensiques dans le cadre d'enquêtes liées à la cybercriminalité. Par exemple, déterminer des plages de données manquantes sur le disque correspondant à des fichiers supprimés, qui pourraient être ensuite récupérés en tant que preuves.

1.2 Présentation du projet

Les fonctionnalités souhaitées par notre client pour cette librairie étaient, entre autres, de pouvoir afficher l'arborescence des fichiers d'un dossier et d'assurer la traçabilité de l'accès à un fichier ainsi que de ses métadonnées, ainsi que de créer un container test pour les systèmes de fichiers analysés.
Pour effectuer une analyse binaire du disque à partir de zéro, il est nécessaire d'analyser les tables de partitions situées au début du disque. Ces tables sont placées dans les premiers octets d'un disque et contiennent la liste des partitions, leurs adresses de début et de fin, et d'autres informations comme leur type ou leurs permissions.
Il faut ensuite accéder aux partitions grâce à l'adresse donnée par la table, où il faudra lire et décoder l'entête qui est présente dans les premiers octets de la plupart des types de partitions. Cette entête dépend du type de partition, mais elle contiendra généralement des informations comme la taille du système de fichiers, la taille de l'unité de base de stockage, et l'espace utilisé et/ou libre.

Pour cela nous avons d'abord besoin de nous renseigner sur les spécifications et les structures des différents types de systèmes de fichiers.

1.2.1 Exemple de structure d'entête

En ext4 par exemple, l'entête est situé dans le premier block et est appelé superblock. Il contient des informations comme le nombre total d'inode et de blocks, le nombre d'inodes et de blocks libres, le nombre de blocks et d'inodes par groupe et les algorithmes de chiffrement utilisés. Ce superblock est stocké dans chaque groupe de blocks pour assurer la fiabilité de la partition.

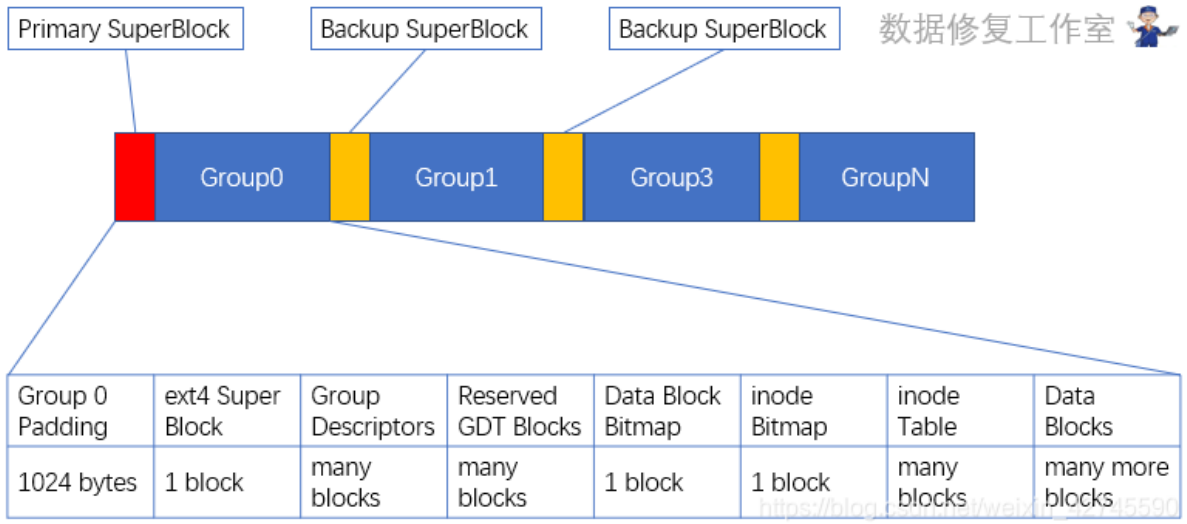

Figure 1: Structure des entêtes de blocks en ext4

Le système de fichier est ensuite divisé en groupes de blocks, dont la structure est présentée ci-dessus.

Le système de fichiers ext4 est basé sur des inodes, qui représentent les fichiers. Ils contiennent toutes les métadatas d'un fichier dont les temps d'accès, de modification et de création, la carte des blocks occupés par le fichier.

Les groupes de blocks contiennent donc une table des inodes qui répertorie tous les inodes stockés dans ce groupe, ainsi que leur adresse. Les bitmaps d'inode et de blocks servent quant à elles à déterminer quels blocks ou inodes sont utilisés et libres. Elles contiennent une séquence de bits correspondant à la séquence de blocks ou inodes stockés dans le groupe de blocks. Un bit à 1 indique un block ou inode occupé, tandis qu'un bit à 0 indique que l'inode ou le block est libre.

Implémenter l'analyse d'une partition implique plusieurs niveaux de complexité. Premièrement, il existe différents types de partition, organisé autour de différentes structures de données et mécanisme de gestion des fichiers et de leur contenu. De plus, le code doit être portable à tout type de système d'exploitation, car chacun aura des contraintes spécifiques pour accéder aux données d'un disque.

2 Présentation de la méthodologie utilisée

Pour ce projet, nous avons appliqué les méthodes d'Agilité en informatique.

Notre projet est utilisable sous 3 angles différents. La bibliothèque en GO sera utile pour des développeurs qui veulent utiliser les fonctions. Elle est facilement utilisable grâce aux appellations de fonctions claires ainsi qu'à la documentation. Elle peut être utilisé aussi dans des projets depuis n'importe quel langage grâce à l'utilisation de l'API. Et enfin, son utilisation est illustrée grâce à l'interface graphique. Notre projet a l'atout de pouvoir être utilisé par différents types d'utilisateurs, du développeur au particulier.

Nous avons réparti les différents étages de l'analyse du disque dans des fichiers différents pour assurer la lisibilité et maintenabilité du projet. Les principes SOLID sont appliqués. En effet, les fonctions sont réparties dans les différents fichiers selon si elles gèrent le disque, les partitions, les fichiers, les blocks ou l'API.

Pour l'interface graphique, nous avons utilisé une architecture Model-View-Presentation qui a permis d'améliorer l'implémentation de cette dernière. De plus, chaque panel de cette interface possède sa propre classe. Cette disposition permet la séparation des fonctions selon chaque outil présent dans l'interface et cela permet d'éviter la surcharge du fichier principal.

3 Présentation des objectifs du projet

Présentation des objectifs du projet (finaux et à mi-parcours) :

3.1 Objectifs à mi-parcours :

Développer une librairie en Go permettant de :

- Lister et récupérer des informations sur les disques présents sur le système
- Lister et récupérer des informations sur les partitions d'un disque choisi
- Lister les fichiers de la partition à partir d'un chemin
- Déterminer les adresses et le contenu des blocks qui correspondent aux données d'un fichier donné

3.2 Objectifs finaux :

- Développer une librairie en Go permettant de (en plus des objectifs à mi-parcours) :
  - Offrir une API permettant d'accéder aux fonctionnalités de la librairie avec des requêtes renvoyant des données formatées en JSON.
  - Déterminer les espaces non alloués sur le disque
  - Fonctionner sur n'importe quel système d'exploitation
- Développer une interface graphique permettant d'accéder à toutes les fonctionnalités de la librairie en utilisant l'API.
- Tester la librairie sur plusieurs systèmes d'exploitation
- Mettre en place un environnement de test avec un conteneur Docker

4 Présentation du travail réalisé et des objectifs atteints :

La librairie que nous présentons remplit la plupart des fonctions que nous souhaitions offrir à l'utilisateur. Le travail que nous avons réalisé consiste en deux parties, le backend qui est la librairie, et une interface graphique permettant de présenter les fonctionnalités.

4.1 Backend

Le backend est divisé en plusieurs parties, chacune implémentée dans des fichiers Go dédiés.
Tout d'abord, il permet de récupérer l'ensemble des disques présents sur le système, qu'ils soient internes à l'ordinateur (port NVME) ou externes (branchés en USB). Les fonctions de récupération des disques sont situées dans le fichier disk.go. Ce fichier inclut aussi la structure DiskInfo, qui permettra par la suite de transmettre les informations au format JSON.

```
type DiskInfo struct {
        Name     string  `json:"name"`
        Capacity float64 `json:"capacity_gb"`
}
```

Figure 2: Structure DiskInfo

À partir d'un disque donné, les fonctions du fichier partition.go permettent ensuite de récupérer la liste des partitions d'un disque donné, à nouveau formaté en JSON pour simplifier la communication par l'API. On peut obtenir toutes les partitions des disques, y compris celles dont le système de fichier n'est pas connu. La structure renvoyée en JSON est PartitionInfo, qui regroupe les principales informations pertinentes sur une partition.

```
type PartitionInfo struct {
        Name         string          `json:"name"`
        FSType       string          `json:"fs_type,omitempty"`
        MountPoint   string          `json:"mount_point,omitempty"`
        Size         string          `json:"size"`
        FSSize       string          `json:"fs_size,omitempty"`
        FSUsed       string          `json:"fs_used,omitempty"`
        UUID         string          `json:"uuid,omitempty"`
        PartTypeName string          `json:"part_type_name,omitempty"`
        PartNumber   json.RawMessage `json:"part_number,omitempty"`
        PhySec       json.RawMessage `json:"physical_sector_size,omitempty"`
        LogSec       json.RawMessage `json:"logical_sector_size,omitempty"`
}
```

Figure 3: Structure PartitionInfo

On peut ensuite lister les fichiers et dossiers situés à un emplacement donné d'une partition, grâce aux fonctions du fichier file.go. Comme les autres fonctionnalités de la librairie, il est possible de récupérer les informations d'un fichier sous JSON, dans la structure FileInfo.

```
type FileInfo struct {
        Name            string `json:"name"`
        Type            string `json:"type"`
        IsSymlink       bool   `json:"is_symlink"`
        SizeBytes       int64  `json:"size_bytes"`
        Permissions     string `json:"permissions"`
        HardLinks       uint64 `json:"hard_links"`
        Inode           uint64 `json:"inode"`
        OwnerUID        uint32 `json:"owner_uid"`
        OwnerGID        uint32 `json:"owner_gid"`
        BlockSize       int64  `json:"block_size"`
        BlocksAllocated int64  `json:"blocks_allocated"`
        LastModified    string `json:"last_modified"`
        LastAccess      string `json:"last_access"`
}
```

Figure 4: Structure FileInfo

Il est ensuite possible d'obtenir la liste des blocks contenant les données un fichier et leur contenu en hexadécimal. Cela permet d'avoir une meilleure visualisation de la structure sous laquelle le fichier est stocké sur le disque. L'affichage des blocks est géré par le fichier block.go, et les résultats sont à nouveau envoyés au format JSON.

Pour plus de simplicité d'utilisation, il est aussi possible d'utiliser un filtre de recherche permettant de sélectionner le type des fichiers, ou une partie de leur nom. Le fichier filter.go permet d'implémenter cette recherche, qui renvoie une structure JSON FilesResponse qui sera celle renvoyée par l'API après une requête.

```
type FilesResponse struct {
        MountPath string     `json:"mount_path"`
        Files     []FileInfo `json:"files"`
}
```

Figure 5: Structure FilesResponse

Le backend remplit ces fonctionnalités :

- Lister et récupérer des informations sur les disques présents sur le système
- Lister et récupérer des informations sur les partitions d'un disque choisi
- Lister les fichiers de la partition à partir d'un chemin
- Déterminer les blocks qui correspondent aux données d'un fichier donné

Pour offrir un cas d'utilisation de notre librairie, nous avons implémenté une API locale à laquelle il est possible d'envoyer des requêtes pour obtenir les informations renvoyées par la librairie dans un format JSON plus simple.

Notre API est un serveur local sur le port 8080, auquel il est possible d'envoyer des requêtes GET pour exploiter les fonctions de la librairie sans forcément l'implémenter en Go. Les endpoints disponibles correspondent aux fonctions de la librairie.

/disks

Cet endpoint renvoie la liste des disques au format JSON spécifié dans la structure DiskInfo (Figure 2).

/partitions

Cet endpoint renvoie la liste des partitions du disque spécifié, au format de la structure PartitionInfo (Figure 3), en prenant comme paramètre `disk` le nom du disque.

/files

Cet endpoint renvoie la liste des fichiers au chemin spécifié, au format de la structure FilesResponse (Figure 5), en prenant comme paramètres `partition` la partition, `path` le chemin dans la partition et `filter` le filtre pour affiner les recherches.

/blocks

Cet endpoint renvoie la liste des blocks composant le fichier situé au chemin spécifié, en prenant comme paramètres `partition` la partition, `path` le chemin menant au fichier dans la partition.

Les problèmes liés à une mauvaise requête ou problème interne au backend sont gérées et renverront le message d'erreur adéquat à la requête.
Pour gérer le serveur local et la réponse aux requêtes, nous avons utilisé la librairie standard du Go ainsi que Gin Web Framework. Cela permet la portabilité de l'API.

```
❯ sudo /usr/local/go/bin/go run ./api.go ./block.go ./disk.go ./filter.go ./file.go ./partition.go ./utils.go
[sudo] password for pentafreeerror:
[GIN-debug] [WARNING] Creating an Engine instance with the Logger and Recovery middleware already attached.

[GIN-debug] [WARNING] Running in "debug" mode. Switch to "release" mode in production.
 - using env:   export GIN_MODE=release
 - using code:  gin.SetMode(gin.ReleaseMode)

[GIN-debug] GET    /disks                    --> main.main.func1 (3 handlers)
[GIN-debug] GET    /partitions               --> main.main.func2 (3 handlers)
[GIN-debug] GET    /files                    --> main.main.func3 (3 handlers)
[GIN-debug] GET    /blocks                   --> main.main.func4 (3 handlers)
[GIN-debug] [WARNING] You trusted all proxies, this is NOT safe. We recommend you to set a value.
Please check https://pkg.go.dev/github.com/gin-gonic/gin#readme-don-t-trust-all-proxies for details.
[GIN-debug] Listening and serving HTTP on 0.0.0.0:8080
[GIN] 2025/03/20 - 19:19:45 | 200 |    6.099193ms |       127.0.0.1 | GET      "/disks"
[GIN] 2025/03/20 - 19:19:46 | 200 |   21.454365ms |       127.0.0.1 | GET      "/partitions?disk=%2Fdev%2Fnvme0n1"
[GIN] 2025/03/20 - 19:19:49 | 200 |    3.946625ms |       127.0.0.1 | GET      "/files?partition=/dev/nvme0n1p6&path="
[GIN] 2025/03/20 - 19:19:59 | 200 |    2.463002ms |       127.0.0.1 | GET      "/files?partition=/dev/nvme0n1p6&path="
[GIN] 2025/03/20 - 19:20:16 | 200 |    3.042728ms |       127.0.0.1 | GET      "/disks"
[GIN] 2025/03/20 - 19:20:17 | 200 |    62.14588ms |       127.0.0.1 | GET      "/partitions?disk=%2Fdev%2Fnvme0n1"
[GIN] 2025/03/20 - 19:20:19 | 200 |    2.370277ms |       127.0.0.1 | GET      "/files?partition=/dev/nvme0n1p6&path="
[GIN] 2025/03/20 - 19:20:30 | 200 |    1.260767ms |       127.0.0.1 | GET      "/files?partition=/dev/nvme0n1p6&path=home"
[GIN] 2025/03/20 - 19:20:31 | 200 |   14.875143ms |       127.0.0.1 | GET      "/files?partition=/dev/nvme0n1p6&path=home/pentafreeerror"
[GIN] 2025/03/20 - 19:20:34 | 200 |    6.249761ms |       127.0.0.1 | GET      "/blocks?partition=%2Fdev%2Fnvme0n1p6&path=home%2Fpentafreeerror%2F.bashrc"
[GIN] 2025/03/20 - 19:33:04 | 200 |    6.271228ms |       127.0.0.1 | GET      "/files?partition=/dev/nvme0n1p3"
```

Figure 6: Appels API vus depuis le terminal

4.2 Interface graphique

Notre librairie est donc accessible depuis 3 niveaux : un logiciel écrit en Go qui appellerait directement les fonctions, un logiciel écrit en n'importe quel langage qui exploiterait l'API pour récupérer les informations, ou l'interface graphique écrite en Python. Cette dernière utilise la bibliothèque grâce à des appels dans une API. Voici une illustration de notre interface graphique :

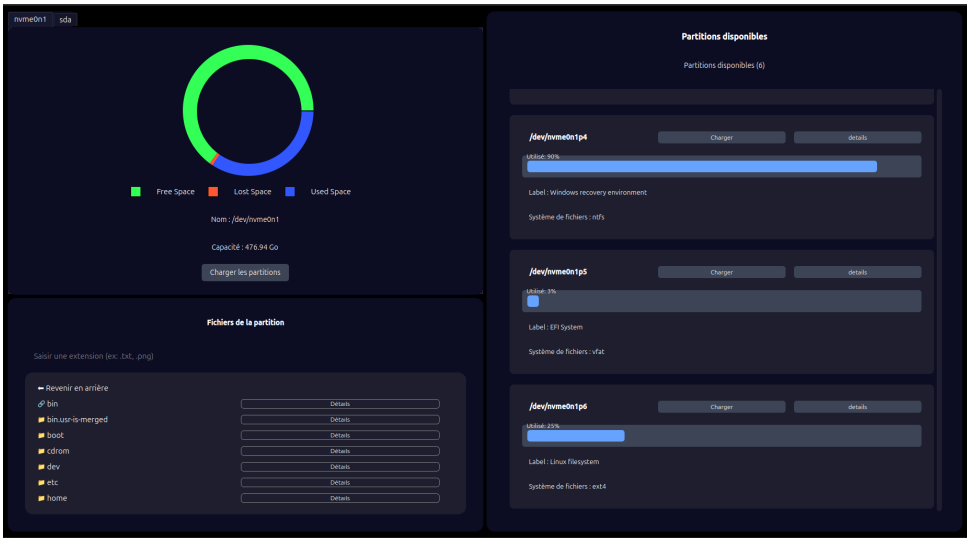

Figure 7: Interface graphique

a) Récupération des disques

Dès l'ouverture de l'interface graphique, un appel à l'API est effectué, cet appel permet de charger les informations sur les disques présents sur la machine. Les différents disques disponibles sont affichés sous forme d'onglets en haut du panneau d'affichage. Dans chaque onglet, est contenu un diagramme qui représente l'espace sur le disque. Il est réparti en trois catégories : l'espace occupé, l'espace libre et l'espace perdu.

Pour chaque disque, il est possible de charger les partitions qui sont présentes dessus. Grâce au bouton "Charger les partitions", l'interface graphique fera un nouvel appel API pour récupérer les partitions associées à ce disque.

b) Affichage des partitions

Une fois les partitions chargées, ces dernières apparaitront sur le panneau de droite. Pour chaque partition, on affiche son nom, son pourcentage d'utilisation, son label ainsi que son système de fichiers. Il est possible d'afficher plus de détails sur cette partition grâce au bouton "details". Pour accéder aux fichiers présents sur une partition, il suffit de cliquer sur le bouton "charger".

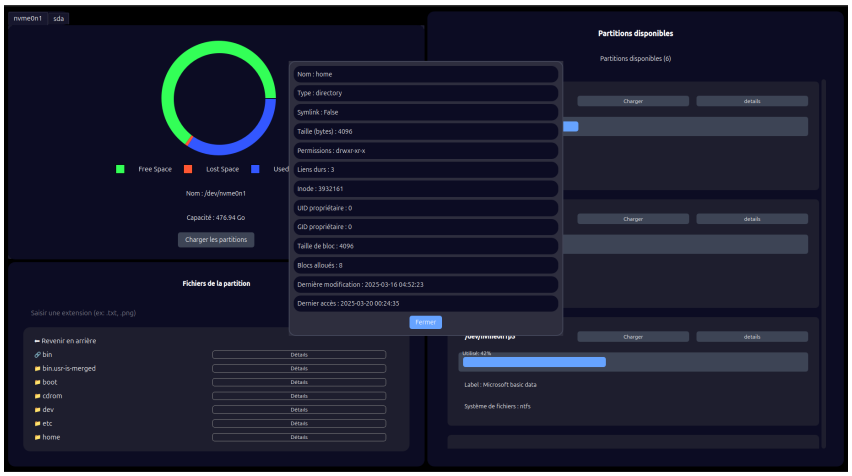

Figure 8: Affichage des détails d'une partition

c) Affichage des fichiers récupérés

Pour accéder aux fichiers, l'interface graphique effectue un troisième appel à l'API pour récupérer les fichiers associés à cette partition. Les fichiers seront affichés dans le tableau sous l'onglet du disque. Dans ce tableau, on retrouve tous les fichiers et dossiers présents à la racine de la partition. Il est possible de se déplacer comme dans un système de gestion de fichiers. Chaque fichier est accompagné d'un bouton "Détails" et d'un bouton "Blocks". Le premier permet d'afficher plus d'informations sur le fichiers pareillement à ce qui est fait pour les partitions. Le second permet d'afficher tous les blocks qui composent ce fichier. En passant la souris sur ces blocks, il est possible de voir le contenu du block.

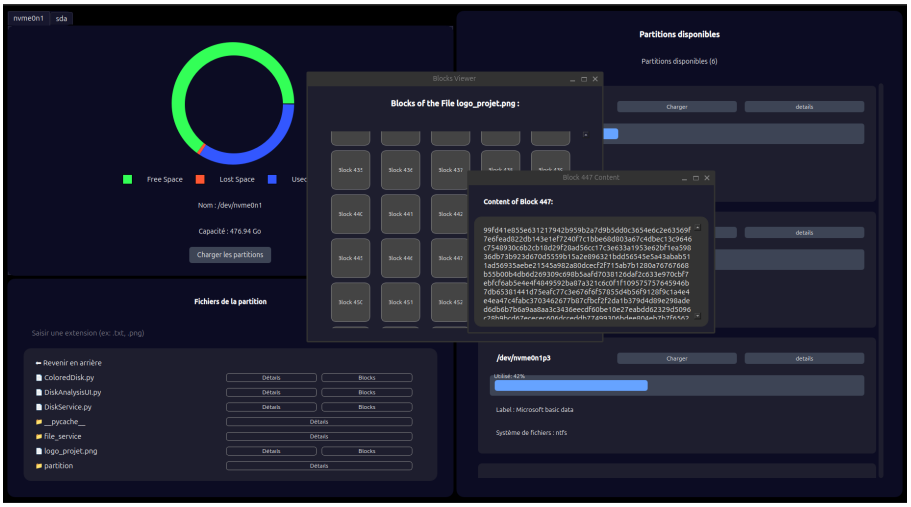

Figure 9: Affichage des blocks d'un fichier

Il est aussi possible de filtrer les fichiers qui s'affichent grâce à une barre de recherche. Il suffit d'indiquer l'extension de fichier que l'on souhaite trouver.

5 Objectifs non atteints

Dans le cadre du projet, plusieurs objectifs n'ont pas pu être atteints en raison de contraintes de temps et de l'acquisition de nouvelles compétences. Voici un résumé des principaux objectifs non réalisés, les raisons de ces manquements et les conséquences sur le développement.

a) Mise en place d'un environnement de tests automatisés avec conteneurisation (Docker)

- **Objectif non atteint** : L'infrastructure de tests automatisés avec Docker n'a pas été mise en place.
- **Motif** : Le temps nécessaire pour configurer Docker et les tests automatisés a été sous-estimé. De plus, l'équipe a dû se former sur Go, un langage relativement nouveau pour nous.
- **Conséquence** : Les tests ont été réalisés uniquement sur des systèmes locaux (clés USB, disques externes), limitant la validation multiplateforme et l'assurance de la stabilité sur divers systèmes.

b) Tests sur d'autres systèmes d'exploitation (Windows, macOS)

- **Objectif non atteint** : Tests réalisés uniquement sur Linux.
- **Motif** : Manque de ressources matérielles pour tester sur Windows et macOS, et manque de temps pour explorer ces plateformes.
- **Conséquence** : L'absence de tests sur ces systèmes crée un risque d'incompatibilité non détecté, limitant l'adoption de l'API dans des environnements variés.

c) Mise en place d'une gestion des erreurs robuste

- **Objectif non atteint** : La gestion des erreurs, notamment pour les partitions corrompues et les permissions, n'a pas été suffisamment développée.
- **Motif** : Le focus a été mis sur les fonctionnalités principales, et la gestion des erreurs a été négligée à cause des contraintes de temps.
- **Conséquence** : L'API offre des messages d'erreur peu détaillés, ce qui complique le diagnostic et l'identification des problèmes, affectant ainsi sa robustesse.

d) Détails sur les Blocks et les Fichiers/Dossiers

- **Objectif non atteint** : Fournir des informations détaillées sur les blocks (adresse exacte) et l'offset des fichiers par rapport au début de la partition.
- **Motif** : Ces fonctionnalités n'ont pas été implémentées à cause du manque de temps. Le calcul précis des adresses des blocks et des offsets aurait nécessité plus de travail.
- **Conséquence** : Cette absence de précision limite la qualité des informations renvoyées par l'API, laissant les utilisateurs sans ces détails avancés pour une gestion optimisée des données.

6 Perspectives d'évolution du projet

Malgré les objectifs non atteints, plusieurs améliorations peuvent être apportées au projet. Voici des pistes pour l'avenir :

a) Mise en place d'une infrastructure de tests conteneurisés (Docker)

- **Proposition** : Automatiser les tests avec Docker sur des environnements multiples (Linux, Windows, macOS) pour garantir la compatibilité et l'intégration continue.
- **Impact** : Cela améliorerait la stabilité du projet et simplifierait sa maintenance, rendant l'API plus fiable sur diverses plateformes.

b) Expansion des tests sur différents systèmes d'exploitation

- **Proposition** : Tester l'API sur Windows et macOS pour s'assurer de son bon fonctionnement sur ces systèmes.
- **Impact** : Cela élargirait l'adoption du projet à un plus grand nombre d'utilisateurs, augmentant sa portée et sa fiabilité.

c) Amélioration de la gestion des erreurs

- **Proposition** : Ajouter une gestion des erreurs détaillée, notamment pour les partitions corrompues et les permissions de fichiers.
- **Impact** : Cela renforcerait l'expérience utilisateur en rendant l'API plus robuste et plus facile à utiliser.

d) Optimisation des performances

- **Proposition** : Optimiser l'API pour gérer efficacement les grandes partitions et les systèmes de fichiers volumineux, en utilisant des techniques comme la pagination ou la mise en cache.
- **Impact** : L'API serait plus rapide et réactive, améliorant ainsi l'expérience utilisateur, en particulier dans les environnements professionnels avec de grandes quantités de données.

e) Approche "From Scratch" pour certaines fonctionnalités avancées

- **Proposition** : Explorer une approche "from scratch" pour les fonctionnalités avancées, en évitant les bibliothèques externes.
- **Impact** : Bien que cela prenne plus de temps, cette approche offrirait un contrôle total sur le projet et permettrait une meilleure gestion des partitions et des blocks de fichiers.

7 Présentation de la répartition des tâches

L'implémentation principale du projet a été effectuée par l'ensemble du groupe, pilotée par Abdulaziz. Nous avons appris à coder en Go pour ce projet. Voici les points supplémentaires sur lesquels les membres du groupe ont travaillé :

- Antony : implémentation de l'API
- Yam : lien entre l'API et l'interface graphique
- Yahya : tests de la bibliothèque
- Abdulaziz : implémentation de l'API
- Maëlys : interface graphique

À la fin de l'implémentation du projet, nous nous sommes regroupés pour améliorer le front-end de l'interface graphique et pour y ajouter des fonctionnalités avancées comme la navigation dans le système de fichiers et le filtre par extension.

8 Bilan sur le travail de groupe

8.1 Ce qui n'a pas marché

Malgré notre motivation et nos efforts pour mener à bien ce projet, plusieurs éléments ont freiné notre progression et complexifié notre organisation.

1. Des environnements de travail hétérogènes

L'un des premiers obstacles que nous avons rencontrés concernait les différences dans nos environnements de développement. Chaque membre du groupe utilisait un système d'exploitation et une distribution différente (Windows, différentes distributions Linux comme Debian 12, Kali Linux et Ubuntu), ce qui a rendu difficile la mise en place d'un environnement de travail standardisé. Comme notre projet impliquait des aspects techniques relativement bas niveau, certaines configurations logicielles ne fonctionnaient pas de la même manière selon l'OS utilisé. Par exemple, certains outils ou bibliothèques n'étaient pas compatibles d'une plateforme à l'autre, ce qui a nécessité du temps supplémentaire pour trouver des alternatives ou adapter le code. De plus, des différences dans la gestion des dépendances et des permissions système ont entraîné des problèmes d'exécution qui n'étaient pas toujours faciles à résoudre.

2. Une communication parfois compliquée

Un autre frein majeur a été la communication au sein du groupe. Nous étions issus de spécialisations différentes (EPCS et Cyber IA), ce qui a parfois entraîné des difficultés dans la communication, ce qui a rallongé la prise de décision.

8.2 Ce qui a bien fonctionné

Pour faire face à ces difficultés, plusieurs stratégies et bonnes pratiques mises en place au fil du projet nous ont permis de progresser efficacement et d'atteindre nos objectifs.

1. Un dépôt Git bien structuré et un workflow organisé

L'un des éléments qui nous a le plus aidés a été la mise en place d'un dépôt Git bien structuré dès le début du projet. Nous avons adopté un workflow clair, basé sur des branches dédiées pour chaque fonctionnalité, des revues de code systématiques et des règles précises pour la gestion des commits. Cette organisation a permis d'éviter de nombreux conflits de fusion et de garantir une meilleure traçabilité des modifications. Chacun pouvait travailler sur sa partie sans risque d'écraser le travail des autres, et les revues de code ont permis d'améliorer la qualité globale du projet en détectant d'éventuelles erreurs ou incohérences avant l'intégration du code dans la branche principale.

2. Un travail de recherche approfondi en amont

Un des points les plus positifs de notre organisation a été le temps que nous avons consacré à la recherche et à la compréhension du sujet avant de nous lancer dans le développement. Plutôt que de coder directement sans réelle préparation, nous avons pris le temps d'étudier les concepts théoriques, d'analyser différentes approches possibles et d'évaluer les meilleures solutions techniques avant d'entamer l'implémentation. Cette phase de documentation et d'analyse nous a permis d'avoir une vision claire des enjeux du projet et d'éviter de nombreuses erreurs qui auraient pu nous ralentir plus tard. Une fois que nous avons abordé la partie technique, le développement s'est fait de manière beaucoup plus fluide et rapide, car nous savions exactement où nous allions et quelles méthodes utiliser. C'est grâce à cela que la répartition des tâches a été faite efficacement en fonction des compétences.

Nous avons également tiré profit des ressources disponibles (documentation officielle, forums spécialisés) pour anticiper d'éventuelles difficultés et trouver des solutions adaptées avant même qu'elles ne se présentent. Cela nous a permis de limiter les pertes de temps liées aux essais-erreurs et d'accélérer notre progression.

3. Une répartition efficace des tâches en fonction des compétences

Nous avons également su tirer parti des compétences de chacun pour répartir efficacement le travail. Plutôt que d'attribuer les tâches de manière aléatoire, nous avons pris en compte les forces et faiblesses de chaque membre pour leur confier des parties du projet où ils étaient le plus à l'aise. Par exemple, ceux ayant une expertise plus poussée en cybersécurité ont pris en charge les aspects liés à la gestion des permissions et à la sécurisation des données. Cette répartition intelligente nous a permis de gagner du temps et d'éviter les blocages liés à un manque de compétences spécifiques sur certaines parties du projet. De plus, les membres plus expérimentés ont pu aider les autres à monter en compétences, ce qui a renforcé la dynamique du groupe et amélioré notre efficacité globale.