Spatial Frinet : application des index spatiaux aux traces d'exécution

Théo Emeriau
theo.emeriau@synacktiv.com

Synacktiv

Résumé. Frinet [4] est un plugin pour *IDA* qui intègre les données de trace d'exécution au désassembleur. Ces traces représentent une mine d'or d'informations à propos du programme cible. La corrélation de ces données permet de traiter efficacement des tâches qui étaient jusque-là manuelles et fastidieuses. Cependant, un défi technique subsiste : en conditions réelles, les traces d'exécution peuvent atteindre et dépasser le milliard d'instructions, tandis que le backend actuel sature aux alentours de 200 millions. Pour passer à l'échelle, un changement de paradigme s'impose : plutôt que de parcourir linéairement la trace, l'état du processus à un instant donné est désormais recherché spatialement, tel un point dans une forêt de rectangles. Cet article détaille la conception de ce nouveau backend de Frinet fondé sur les Packed Hilbert R-Tree.

1 Introduction

La rétro-ingénierie repose traditionnellement sur la complémentarité de deux approches : l'analyse statique et l'analyse dynamique. Ces deux méthodes permettent d'obtenir des informations de natures différentes sur une même cible. Historiquement, les outils de rétro-ingénierie se sont spécialisés d'un côté ou de l'autre ; en conséquence, la corrélation des données reste encore aujourd'hui une tâche principalement manuelle.

Récemment, la fusion de ces informations au sein d'un outil unique est apparue comme une piste prometteuse. Cependant, l'implémentation reste complexe en raison de l'hétérogénéité des données. Elle nécessite non seulement des avancées techniques, mais aussi le développement de nouveaux modes d'interaction et de visualisation.

En 2021, Markus Gaasedelen a publié un plugin pour le décompilateur IDA nommé Tenet [1]. Son objectif est d'intégrer les informations d'une trace d'exécution directement dans l'interface du désassembleur. Grâce à Tenet, IDA combine la puissance d'un décompilateur et d'un *Time Travel Debugger*.

En 2023, Synacktiv a publié Frinet : un fork de Tenet avec de nouvelles fonctionnalités et un traceur basé sur l'outil d'instrumentation dynamique

Frida. Régulièrement utilisé au sein du pôle reverse-engineering de Synacktiv, l'outil excelle sur certaines tâches, notamment :

— Suivre l'évolution du contenu d'un buffer durant un pipeline de transformation.
— Localiser le code qui a émis un log spécifique en recherchant la construction du message dans la mémoire.

Toutefois, les traces acquises en conditions réelles sont de l'ordre de 100 millions à 2 milliards d'instructions. Exploiter cette quantité de données devient critique à mesure que la taille des traces augmente. Actuellement, au-delà d'une limite située aux alentours de 200 millions d'instructions, le manque de fluidité de l'affichage et des interactions rend l'utilisation de Frinet laborieuse, voire impossible sur certaines cibles.

L'objectif de ces travaux est de réécrire le backend de Frinet en optimisant la latence de récupération d'informations dans la trace, et plus précisément, garantir une latence inférieure à 1 milliseconde par recherche. Cette limite correspond au budget temporel par requête dans le scénario suivant : chaque rendu graphique nécessite 15 requêtes, avec un taux de rafraîchissement de 60 images par seconde.

Pour y parvenir, l'idée principale est de restructurer le contenu de la trace d'exécution sous la forme d'un ensemble d'arbres équilibrés multidimensionnels. Ce changement de perspective permet de supporter des traces d'exécution bien plus conséquentes, car la hauteur de l'arbre croît logarithmiquement avec le nombre d'instructions.

2 Trace d'exécution

L'acquisition d'une trace d'exécution dépend de l'architecture du processeur et parfois de contraintes spécifiques au programme ciblé. Le tracing est un sujet complexe avec ses propres défis que nous ne traiterons pas ici. De plus, la suite de l'article fait abstraction de la méthode d'acquisition et du format de stockage de la trace, dès lors qu'elle correspond à la définition suivante.

Une trace d'exécution est une structure qui associe à chaque unité de temps T_x la séquence d'effets produits par l'exécution de la x-ième instruction assembleur d'un processus. Autrement dit, produire une trace consiste à suivre l'exécution d'un programme en notant les différences entre l'état avant et après l'exécution de chaque instruction.

Ci-dessous un extrait du début d'une trace produite par le traceur de Frinet. Chaque ligne représente une unité de temps et sa séquence d'effets, séparés par une virgule.

```
rip=0x7fdf256e2483,rax=0x7fdf24193009,rdx=0x3
rip=0x7fdf256e2494,mr=0x7ffc857caeb8:f01d2924df7f0000
rip=0x7fdf256e24a5,mw=0x7ffc857caeb8:0000000000000000
rip=0x7fdf256e24b0
...
```

Chaque effet s'interprète ainsi :

— `rip=<valeur>` : écriture d'un registre
— `mr=<adresse>:<octets lus>` : lecture mémoire
— `mw=<adresse>:<octets écrits>` : écriture mémoire

Pour obtenir l'état du processus à l'instant T_x, il faut exécuter virtuellement les effets un à un des x premières instructions. Par définition, l'état du processus à l'instant T_0 est "vide" : les valeurs des registres ainsi que l'ensemble de l'espace d'adressage virtuel sont indéfinis.

2.1 Observations mémoire

Dans l'exemple de trace ci-dessus, tracer les lectures mémoire semble inutile car ces dernières n'altèrent pas l'état du processus. En pratique, elles sont pourtant essentielles dans ces deux cas :

— **Tracing différé** : il est fréquent de ne débuter le tracing qu'à partir d'un point d'intérêt (par exemple, l'entrée d'une fonction spécifique). En effet, tracer un processus dès sa création n'est pas toujours pertinent, ni même techniquement possible. Dès lors, l'état initial des régions mémoire affectées avant le début du tracing est perdu.
— **Écritures externes** : les systèmes modernes utilisent des mécanismes de partage de mémoire entre processus ou composants. Ces régions mémoire accessibles dans le processus tracé peuvent être modifiées de l'extérieur à tout instant. La majorité des traceurs ne sont pas capables de détecter ce type d'écriture.

Ainsi, tracer les lectures mémoire permet de récupérer l'état de ces régions mémoire à la lecture. Malheureusement, ce n'est pas une solution parfaite : la récupération d'information est souvent partielle et la provenance est perdue.

Ce modèle implique un changement de sémantique subtil pour les effets des lectures mémoire. En effet durant l'exécution virtuelle, une lecture est traitée comme une écriture si les octets lus diffèrent de l'état connu de la mémoire. Pour lever l'ambiguïté du terme "lecture", lors du parsing, une transformation est appliquée. La lecture de la trace produit une séquence de lectures mémoire et une séquence d'écritures mémoire qui deviennent, après

transformation, une séquence d'observations mémoire et une séquence d'accès mémoire. Contrairement aux observations mémoire qui contiennent la valeur des octets observés, les accès mémoire ne contiennent que la plage d'adresses.

— Chaque écriture devient systématiquement une observation.
— Chaque lecture devient un accès et également, si nécessaire, une observation.

3 Frinet

Le projet Frinet est découpé classiquement en deux parties : le frontend et le backend. Le frontend prend la forme d'un plugin *IDA*. Son rôle est d'intégrer les informations issues de la trace d'exécution dans l'interface graphique d'*IDA*. Il obtient toutes les données nécessaires en interrogeant le backend.

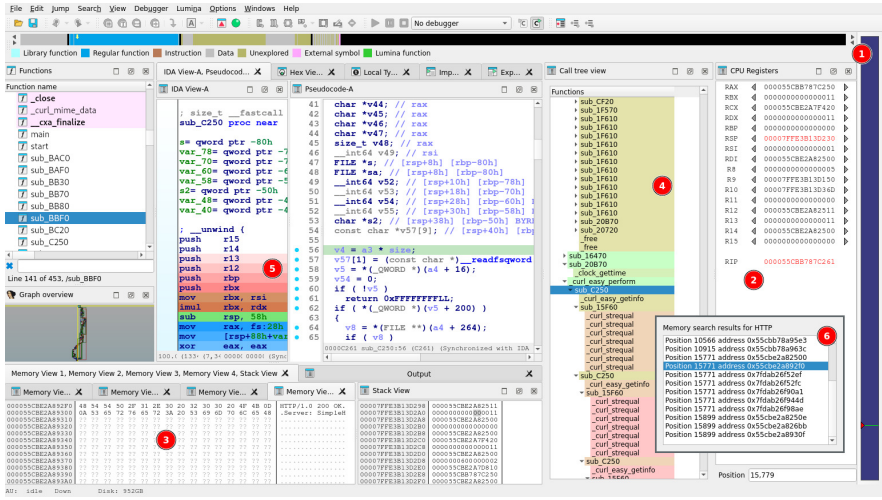

Fig. 1. Capture d'écran du décompilateur *IDA* avec Frinet

Les différents éléments de l'interface de Frinet sont :

1. La timeline complète, où le marqueur rouge indique **la position actuelle** dans la trace d'exécution. Tous les autres éléments sont synchronisés sur ce marqueur.
2. Les valeurs des registres avec une indication visuelle en rouge si le registre vient juste d'être modifié. Autour de chaque valeur, les

flèches gauche/droite permettent de naviguer respectivement vers la précédente et la prochaine mise à jour de ce registre.

3. La vue du contenu d'une région de la mémoire à la position actuelle.
4. La cascade des appels de fonctions obtenue en croisant les valeurs de `PC` avec les fonctions identifiées par *IDA*.
5. L'instruction courante est surlignée en vert, les précédentes en dégradé de rouge et les suivantes en dégradé de bleu. Cette fonctionnalité est particulièrement utile aux frontières des basic blocks pour visualiser rapidement le flux d'exécution entrant et sortant.
6. Les résultats d'une recherche par mot clé à travers l'historique complet de la mémoire.

Le rôle du backend est similaire à celui d'une base de données. Il a accès à la trace d'exécution et doit répondre aux requêtes du frontend. On peut regrouper les requêtes en deux catégories :

— Les requêtes portant sur une partie de l'état du processus à un instant donné, exemple : lister les valeurs des registres ou d'une portion de la mémoire virtuelle à l'instant T.
— Les recherches d'instants vérifiant une condition spécifique, exemple : lister les instants où le registre X3 a été modifié ou trouver le prochain instant T où une adresse mémoire spécifique a été modifiée à partir de l'instant T.

C'est ici que le choix de la structure de données et de l'algorithme de recherche est déterminant. Pour garantir une latence inférieure à une milliseconde, il faut que la méthode choisie soit capable de respecter cette limite dans les pires scénarios possibles : les cas pathologiques.

Pour illustrer ce concept, prenons pour exemple : la recherche de la valeur du registre X12 à T_x. La seule méthode disponible est de parcourir la trace en arrière en partant de T_x jusqu'à trouver une écriture de X12, si elle existe. Les cas où X12 n'a jamais été écrit avant T_x sont pathologiques car ils nécessitent de parcourir les x premières unités de temps avant de pouvoir conclure que la valeur de X12 est indéfinie à T_x. De plus, on remarque que le nombre d'unités de temps à parcourir dans ce scénario croît linéairement avec x, et s'aggrave donc avec la taille de la trace.

Dans cet exemple, le problème n'est pas spécifiquement l'algorithme en lui-même, mais plutôt de ne pas avoir utilisé d'index. Comme les bases de données classiques, il faut construire un index pour répondre efficacement aux requêtes.

Un index est une structure de données qui a pour but d'aider à localiser rapidement les informations. La construction d'un index est coûteuse mais

étant donné que la trace est immuable, le coût est rapidement amorti par les gains de performance.

4 Index par point de sauvegarde

L'index actuel de Frinet, hérité de Tenet, se base sur un système de points de sauvegarde. La construction d'un tel index consiste à exécuter virtuellement la trace d'exécution et à sauvegarder périodiquement l'état du processus.

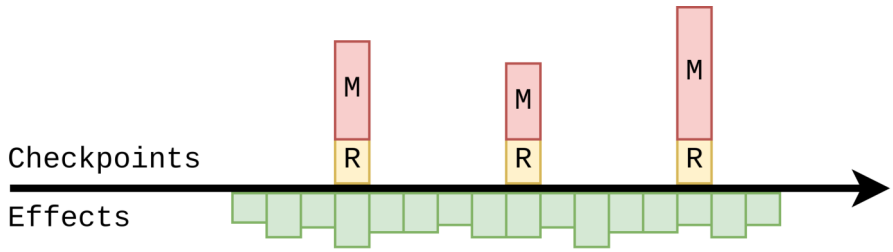

Fig. 2. Schéma d'un index par point de sauvegarde

Chaque point de sauvegarde contient deux sections : les valeurs des registres et la liste des adresses mémoire affectées depuis le dernier point de sauvegarde, annotées respectivement `R` et `M`.

Cet index réduit la complexité des requêtes du premier type dans les cas pathologiques. Pour rechercher la valeur d'un registre à T, il est seulement nécessaire de parcourir en arrière jusqu'au dernier point de sauvegarde. Pour rechercher la valeur d'une adresse mémoire à T, il suffit de trouver le dernier segment qui a affecté cette adresse et de parcourir ce dernier entièrement.

Cette méthode d'indexation a plusieurs problèmes, notamment :

— L'utilité de cet index nécessite de créer des points de sauvegarde très fréquemment. Or, pour une fréquence donnée, le nombre de points de sauvegarde croît linéairement avec la taille de la trace. Utiliser cet index sur des traces d'exécution massives nécessite une quantité de stockage non négligeable.

— Le système de point de sauvegarde ne réduit pas la complexité du deuxième type de requêtes. Par exemple, pour trouver la prochaine écriture du registre X12 à partir de T dans le pire des cas, il faut traverser la trace en partant de T jusqu'à la fin.

— Chaque adresse mémoire doit être recherchée séparément : pour afficher les valeurs d'une plage de 1024 octets à T, il faut faire 1024 requêtes. Cette approche est sous-optimale car, généralement, les observations mémoire affectent une séquence d'adresses.

5 Packed Hilbert R-Tree

Le nouveau backend de Frinet repose sur un ensemble de Packed Hilbert R-Tree [2] pour indexer les traces d'exécution. Le choix de cette structure de données est motivé par la combinaison des propriétés suivantes :

— **Multidimensionnalité** : le R-Tree indexe les objets sur deux dimensions nativement, il n'est donc pas nécessaire d'en préférer une par rapport à l'autre. De plus, les recherches opérant simultanément sur ces deux dimensions, il devient possible d'exploiter les relations entre la topologie des valeurs de chaque axe pour localiser rapidement l'information recherchée.

— **Bounding boxes** : les objets indexés ne sont pas de simples points en deux dimensions mais possèdent également une aire. Cette caractéristique est très utile pour regrouper les octets en mémoire, souvent manipulés comme un seul élément.

— **Stockage à plat** : une fois construit, le R-Tree est composé de listes de structures uniformes, cela permet un chargement quasi instantané en mappant directement le fichier produit en mémoire.

— **Bulk-loading** : il existe un algorithme, nommé "Bulk-loading", pour construire efficacement un R-Tree en une seule passe à partir de la liste des feuilles.

Trois types d'informations sont indexés à l'aide de ces arbres : les observations mémoire, les accès mémoire et les mises à jour des registres. L'index global est ainsi composé de trois catégories d'arbres, chacune étant spécialisée dans un type de donnée.

Au lieu d'être centralisés au sein d'un arbre unique, les éléments de chaque catégorie sont distribués parmi plusieurs R-Tree. Concernant les observations et les accès mémoire, l'espace d'adressage virtuel est fractionné en plusieurs zones, chacune disposant de son propre arbre indépendant. S'agissant des registres, un arbre spécifique est attribué à chacun d'eux. Au final, chaque type d'information est indexé par une forêt de R-Tree. Les justifications de chaque répartition sont détaillées dans les sections suivantes, consacrées à l'application des R-Tree à la mémoire et aux registres.

5.1 Construction d'un R-Tree

Dans un premier temps, nous allons faire abstraction du contexte pour présenter la construction générale d'un Packed Hilbert R-Tree ainsi que l'algorithme utilisé pour les recherches. Les spécificités de chaque type d'arbre sont détaillées dans les sections suivantes.

Le R-Tree (le "R" correspond à "Rectangle") est un arbre où chaque nœud est un rectangle qui englobe ceux de ses enfants. Par transitivité, le rectangle de chaque nœud englobe tous ses enfants récursivement. Il existe plusieurs méthodes pour construire un R-Tree, le terme "Packed" fait référence à la construction "Bulk-loading". Le terme "Hilbert" précise la courbe de remplissage d'espace utilisée lors de la construction.

La première étape est de construire la liste des feuilles, une par objet que l'on souhaite indexer (en rouge sur la figure 3). Les trois types de R-Tree utilisés dans l'index étant de dimension 2, chaque feuille est représentée par un rectangle.

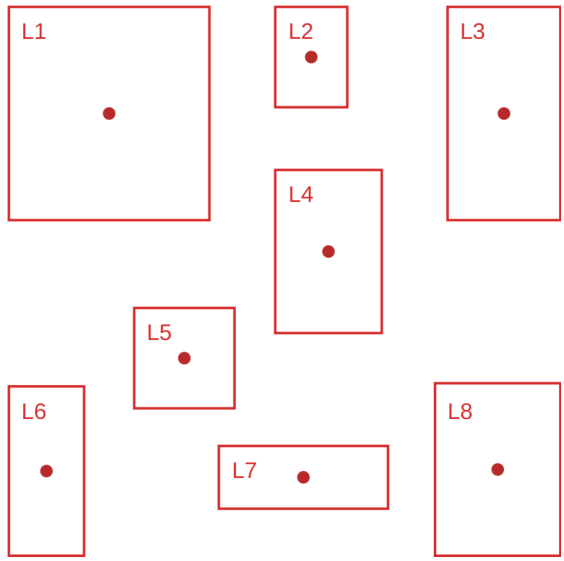

Fig. 3. Construction d'un R-Tree : les feuilles

La deuxième étape consiste à placer toutes les feuilles dans un tableau unidimensionnel de manière à préserver la localité spatiale : deux feuilles voisines dans le plan (2D) doivent idéalement l'être également dans le tableau linéaire (1D). Ce tableau est construit à l'aide d'une courbe de remplissage.

Une courbe de remplissage est une courbe continue qui passe par chaque point d'un carré de côté N. Chaque point se voit associer la distance à parcourir en suivant la courbe pour l'atteindre. Parmi les courbes de remplissage qui préservent la localité spatiale, on trouve notamment la courbe de Hilbert [5] et la courbe de Lebesgue (ou Z-curve). Ces deux courbes sont des fractales : leur construction repose sur un motif auto-similaire qui permet de couvrir intégralement la surface du carré.

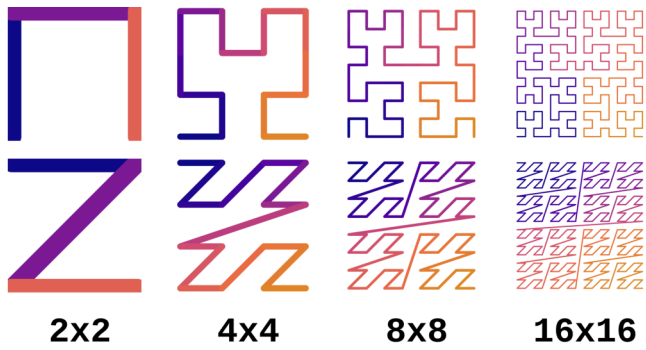

Fig. 4. courbe de Hilbert (en haut) / courbe de Lebesgue (en bas)

La distance de chaque point sur la courbe de Lebesgue est très simple à calculer : il suffit d'entrelacer les bits des deux coordonnées du point. Le même calcul sur la courbe de Hilbert est plus coûteux, mais préserve mieux la localité spatiale. La courbe de Hilbert a été retenue car ce coût de calcul supplémentaire à la construction est largement amorti par l'obtention d'un index plus performant lors de l'exploitation de la trace. La figure 5 illustre le tri des feuilles selon la distance de leur centre sur la courbe de Hilbert.

La dernière étape consiste à construire les niveaux intermédiaires de l'arbre. Pour cela, il faut choisir une taille de groupe. Ce paramètre doit être adapté en fonction de la nature des données et du type de requêtes afin d'optimiser les performances. À titre d'exemple, considérons ici une taille de groupe de 2. Chaque niveau est construit en groupant les X premiers rectangles du niveau précédent. Chaque groupe devient les enfants de ce nœud, le rectangle associé est le rectangle minimal qui englobe tous ses enfants. La construction s'arrête lorsqu'il n'y a plus assez de rectangles pour faire plus d'un groupe (illustré par la figure 6).

En replaçant tous les nœuds dans le plan d'origine, on obtient le R-Tree complet (illustré par la figure 7).

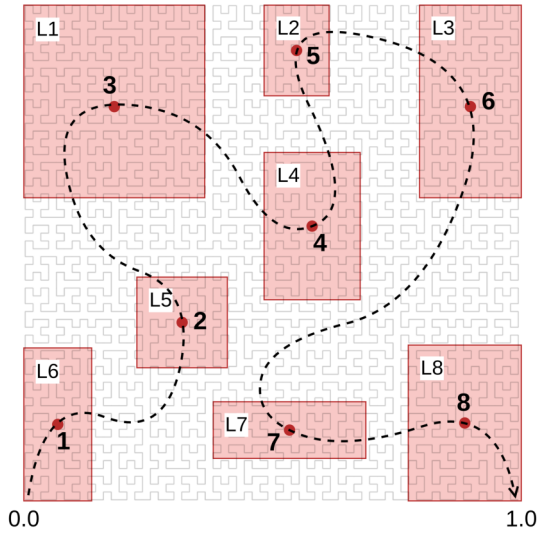

Fig. 5. Construction d'un R-Tree : la courbe de Hilbert

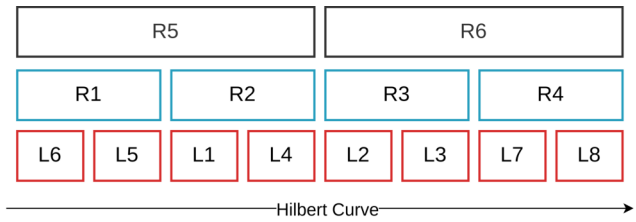

Fig. 6. Construction d'un R-Tree : nœuds intermédiaires

5.2 Requêter un R-Tree

Les R-Tree sont optimisés pour rechercher les feuilles qui intersectent un rectangle R donné. La recherche est un parcours d'arbre en profondeur, au cours duquel les enfants des nœuds n'intersectant pas R sont ignorés. La figure 8 illustre la recherche des feuilles qui intersectent la région verte.

Le parcours de l'arbre commence par les nœuds situés à son sommet. Les rectangles `R5` et `R6` intersectent la région verte donc seulement leurs enfants sont considérés. Parmi les enfants de `R6`, seulement `R4` intersecte la région verte donc les enfants des autres nœuds sont ignorés. Et ainsi de suite jusqu'à atteindre les feuilles de l'arbre qui intersectent la région verte, ici `L7` et `L8`.

Grâce au R-Tree, seulement une partie des feuilles a dû être considérée durant la recherche.

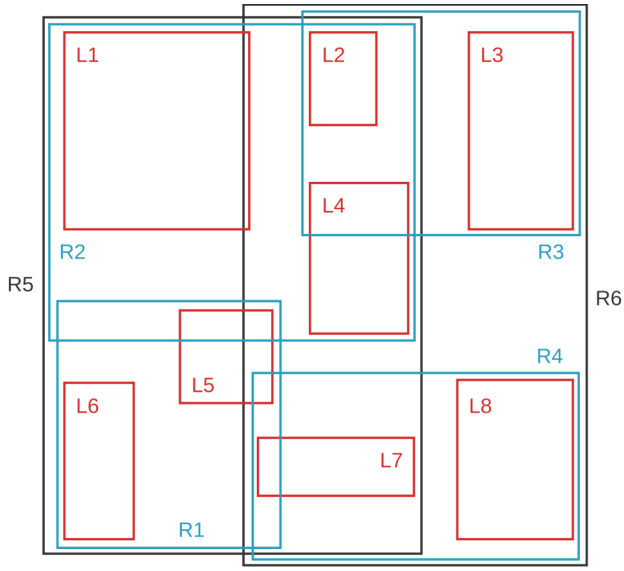

Fig. 7. Construction d'un R-Tree : arbre complet

6 Application des R-Tree à la mémoire

Pour chaque type d'information, l'application des R-Tree nécessite de résoudre ces deux problèmes :

- **Transformation en feuilles** : chaque objet à indexer doit être converti en un ensemble de rectangles (avec ou sans métadonnées).
- **Traduction des requêtes** : pour toute requête visant à identifier les objets vérifiant un prédicat donné, il faut définir le rectangle de recherche R intersectant l'ensemble des feuilles correspondantes.

6.1 Zones mémoire

L'exécution d'un processus s'effectuant dans un espace d'adressage virtuel, seule une infime proportion de cet espace est effectivement utilisée lors d'une exécution donnée. En pratique, les observations et les accès mémoire sont dispersés en îlots (heap, stack, ...) séparés par de vastes zones vides. En conséquence, si l'on utilisait une seule courbe de Hilbert sur le domaine total, cette dernière parcourrait majoritairement du vide, ce qui entraînerait une perte importante de résolution lors de la linéarisation des feuilles. Sans cette segmentation en zones, la perte de résolution provoque une dégradation des performances du R-Tree dès 50 millions d'instructions.

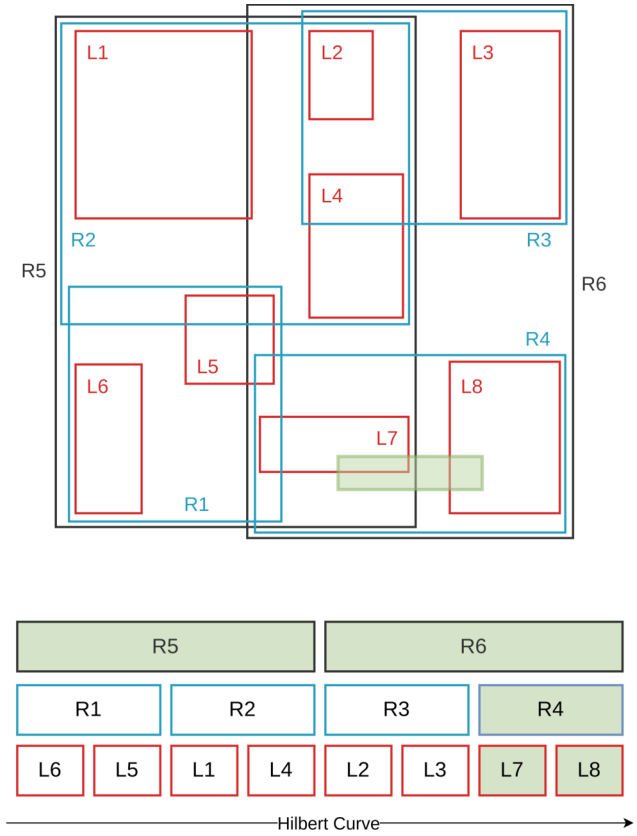

Fig. 8. Recherche dans un R-Tree

Pour pallier ce problème, chaque type d'événement mémoire regroupe ses objets par zones, chacune disposant de son propre R-Tree. Lors de la linéarisation, chaque arbre applique temporairement une normalisation locale des coordonnées de ses feuilles afin d'exploiter la totalité du domaine de sa propre courbe de Hilbert.

La figure 9 illustre un découpage en zones (aux proportions volontairement exagérées) séparant la heap à gauche et la stack à droite.

6.2 Observations mémoire : fragmentation et données brutes

La majorité des instructions assembleur qui affectent la mémoire opèrent sur plusieurs octets simultanément (généralement 4, 8 ou 16), et ces plages d'adresses sont souvent alignées sur leur taille. Ce type d'écriture mémoire est si fréquent qu'il est souhaitable d'encoder ces

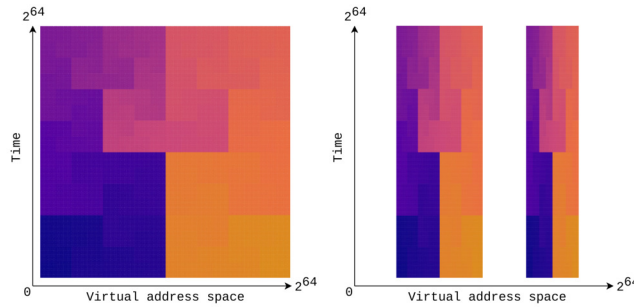

Fig. 9. Zones mémoire

séquences d'octets sous la forme d'une seule feuille, plutôt que de créer une feuille par octet. Cependant, en transposant naïvement chaque observation mémoire comme un rectangle dans le plan, on obtient régulièrement des rectangles qui se chevauchent. Un même point (x,y) pourrait ainsi être couvert par plusieurs rectangles.

Pour illustrer cette problématique, prenons comme exemple cette trace d'exécution fictive de 4 instants :

— T_0 : observation de 2 octets `DA 15` à l'adresse `0x00`
— T_1 : observation de 2 octets `01 62` à l'adresse `0x02`
— T_2 : observation de 2 octets `C5 36` à l'adresse `0x00`
— T_3 : observation de 3 octets `5F 01 8D` à l'adresse `0x00`

La figure 10 illustre le placement une à une des observations mémoire : chaque nouveau rectangle est teinté en vert et le chevauchement de deux feuilles en rouge vif.

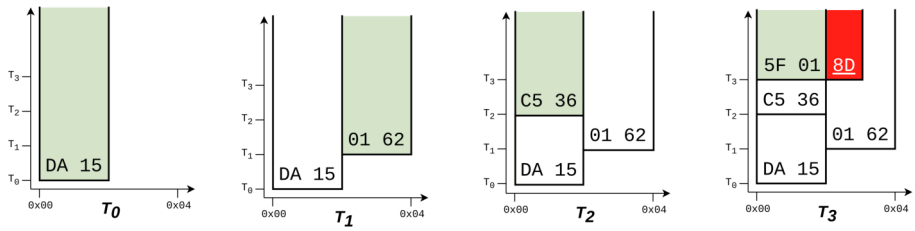

Fig. 10. Chevauchement des observations mémoire

Ce chevauchement entraîne plusieurs effets négatifs. Premièrement, si tous les rectangles étaient disjoints, la recherche de l'unique rectangle couvrant un point donné pourrait être interrompue dès le premier résultat

identifié. Alors qu'avec des rectangles qui se chevauchent, il est nécessaire de poursuivre la recherche pour sélectionner le rectangle le plus récent parmi la liste finale. Deuxièmement, le chevauchement des feuilles accroît mécaniquement le taux de recouvrement des nœuds parents, ce qui dégrade considérablement la qualité de l'index.

Pour contourner ce problème, nous utilisons l'algorithme de fragmentation décrit dans l'article de recherche : NIR-Tree [3]. Lors de l'insertion, chaque nouveau rectangle fragmente les rectangles existants pour lui laisser de la place. Ainsi, tout point (x, y) du plan est couvert par un rectangle au maximum. Si un rectangle contient ce point, alors cet octet a une valeur connue à T_y qui peut être récupérée dans les données brutes. Dans le cas contraire, la valeur de cet octet est indéfinie à T_y.

La figure 11 illustre chaque étape de construction des feuilles disjointes : chaque nouveau rectangle est teinté en vert et les fragments sont teintés en rouge.

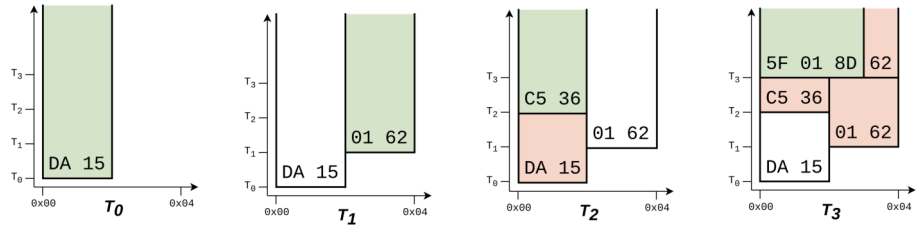

Fig. 11. Fragmentation des observations mémoire

Après fragmentation, les feuilles sont stockées au sein d'un unique tableau contigu : leur structure doit donc être uniforme et la plus compacte possible. Si la feuille est une observation de 8 octets ou moins, les octets sont stockés directement à l'intérieur de la structure. Dans le cas contraire, les octets sont ajoutés à la fin du tableau `rawdata` puis référencés par un offset. Les ajouts précédents dans le tableau des données brutes sont placés dans une hashmap pour dédupliquer les prochaines insertions d'une suite d'octets déjà insérée.

Cette méthode de stockage est intéressante car elle interagit bien avec l'algorithme de fragmentation : chaque fragment d'une feuille est par définition moins large que la feuille originale, donc si le nombre d'octets devient inférieur ou égal à 8, les données sont inline dans le fragment, sinon l'offset est seulement décalé pour pointer vers la sous-partie correspondante.

```
/// Feuille d'un arbre
struct MemLeaf {
    addr_range: (u64, u64),
    time_range: (u32, u32),
    data: Data
}

/// Valeur des octets observés
union Data {
    RawDataOffset(u64),
    Inline([u8; 8])
}
```

6.3 Le cas simplifié des accès mémoire

Contrairement aux observations, la transformation des accès mémoire en feuilles s'avère beaucoup plus simple. Puisqu'un accès représente uniquement un événement temporel, il n'est pas nécessaire d'y associer la valeur des octets accédés. Par conséquent, le chevauchement des feuilles ne pose ici aucun problème d'ambiguïté lors des recherches. Chaque accès mémoire est ainsi directement traduit par un simple rectangle défini par :

— La plage d'adresses accédées allant de `0x0` à `u64::MAX` (axe des abscisses).
— L'instant de l'accès allant de T_0 à T_{max} (axe des ordonnées).

6.4 Traduction des requêtes

Les R-Tree sont optimisés pour trouver les feuilles qui intersectent un point P ou un rectangle R. Par conséquent, il faut traduire les requêtes du frontend en requête d'intersection. La figure 12 illustre la traduction de trois exemples de requêtes.

— Dans l'exemple (a), on veut obtenir les rectangles qui contribuent aux valeurs des octets d'une plage d'adresses à l'instant T_x. La liste des valeurs est reconstituée en concaténant les données brutes des rectangles obtenus.
— Dans l'exemple (b), on recherche l'ensemble des instants où une plage d'adresses a été modifiée et optionnellement ses différentes valeurs au cours du temps.
— Dans l'exemple (c), on recherche parmi la liste des instants où l'adresse `0x2` a été modifiée, les deux instants qui encadrent T_2.

Fig. 12. Illustration de 3 recherches différentes (mémoire)

7 Application des R-Tree aux registres

En ce qui concerne les registres, le backend doit répondre à deux types de requêtes : récupérer la valeur d'un registre à un instant T et trouver l'instant précédent ou suivant par rapport à T où un registre vaut X. Ce second type de requête est primordial pour le registre PC, car il permet au frontend d'implémenter "exécution précédente/suivante".

L'approche la plus simple consisterait à indexer les écritures de tous les registres au sein d'un seul et même R-Tree. L'avantage principal de cette méthode est de rendre possible la récupération des valeurs de tous les registres à un instant T avec une unique recherche. En contrepartie, le second type de recherche serait fortement impacté : cette approche empêche la spécialisation du calcul de la courbe de Hilbert en fonction de la topologie des valeurs propres à chaque registre. En effet, les registres dits “généraux” servent à stocker une grande variété de valeurs (compteurs, pointeurs, données, ...) tandis que le registre PC pointe uniquement sur des régions mémoire contenant du code.

La solution retenue consiste donc à indexer chaque registre dans un R-Tree indépendant, afin de pouvoir améliorer la résolution de la courbe de Hilbert en fonction de la plage des valeurs qu'il atteint au cours de la trace. Chaque écriture de registre est ainsi traduite par un rectangle défini par :

— La valeur écrite dans le registre, allant de `0x0` à `u64::MAX` (axe des abscisses).
— La plage d'instants durant laquelle le registre reste inchangé, allant de T_0 à T_{max} (axe des ordonnées).

De la même manière que pour la mémoire, les requêtes portant sur les registres sont également traduites en requêtes d'intersection. La figure 13 illustre la traduction de trois exemples de requêtes.

— Dans l'exemple (a), on recherche le rectangle contenant la valeur du registre RIP à l'instant T_x.
— Dans l'exemple (b), on identifie les instants où RIP entre ou sort d'une plage d'adresses de code.
— Dans l'exemple (c), on recherche les mises à jour précédente et suivante du registre RSP, correspondant à la plage temporelle du rectangle à l'instant T_x.

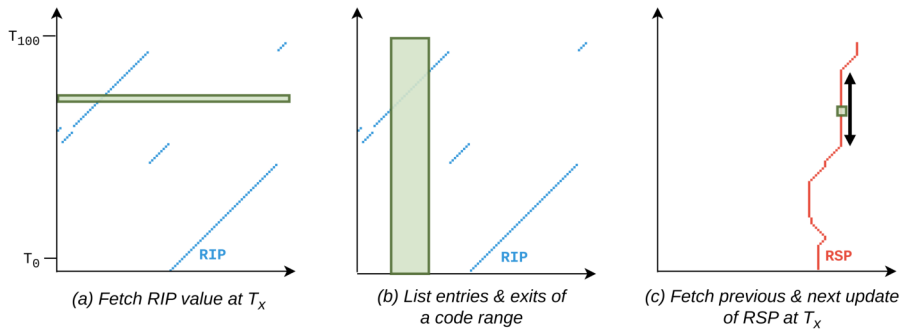

Fig. 13. Illustration de 3 recherches différentes (registres)

8 Défis d'implémentation et passage à l'échelle

L'objectif de ce nouveau backend est de garantir une faible latence pour chaque requête. Cet objectif est atteint grâce à l'index décrit précédemment. Cependant, l'étape de linéarisation présentée dans la section précédente n'est pas utilisable telle quelle, car elle nécessiterait d'être capable de stocker toutes les feuilles simultanément en RAM. Pour une trace d'exécution de plus d'1 milliard d'instructions, cela nécessiterait une quantité de mémoire vive qui dépasse largement les capacités classiques d'un ordinateur portable.

Une adaptation de l'algorithme de construction classique bulk-loading s'impose pour indexer des traces d'exécution massives. Non seulement le nouvel algorithme doit consommer peu de RAM mais surtout, sa complexité temporelle doit être la plus linéaire possible par rapport à la taille de la trace. Vérifier cette contrainte permet de garantir la non-explosion du temps de calcul nécessaire pour construire l'index.

L'algorithme retenu remplace la construction en une seule lecture de la trace, par une construction en quatre phases dont trois lectures complètes de la trace.

La première phase parcourt la trace en collectant les informations nécessaires pour les phases suivantes : le nombre et la taille de chaque zone mémoire, le nombre et la quantité de feuilles de chaque R-Tree, la taille du fichier d'index à pré-allouer, ... Le fichier d'index est pré-alloué dès la fin de la première phase pour simplifier les prochaines phases qui y accéderont via un mapping mémoire du fichier.

La linéarisation est répartie entre la deuxième et la troisième phase. La deuxième phase parcourt la trace et partitionne chaque R-Tree en une grille de 16 par 16, chaque feuille est assignée à la case qui correspond à la distance de son centre sur la courbe de Hilbert en basse résolution (d'ordre 4). Lors de cette phase, seulement le nombre de feuilles assignées à chaque case est collecté.

La troisième phase parcourt la trace et dès que toutes les feuilles d'une case sont disponibles : elles sont ordonnées par la distance sur la courbe de Hilbert haute résolution et stockées dans la région correspondante du fichier d'index mappé en mémoire. Cette méthode divise la quantité de mémoire vive nécessaire par 16, car lors du parcours de la trace au plus les feuilles de 16 cases sur 256 sont stockées en RAM.

La dernière phase ne parcourt pas la trace, elle calcule les nœuds intermédiaires de chaque R-Tree et termine l'écriture des headers et metadata dans l'index.

Après avoir construit l'index sur disque, Frinet doit le charger avant de l'utiliser. Le chargement en mémoire est dit "zero-copy" : il se résume à mapper le fichier en mémoire via *mmap* et à instancier les structures (pointeur et taille) référençant directement cette mémoire. L'opération est donc instantanée, exception faite des *page faults* gérés par le noyau lors de l'utilisation de l'index. Cette méthode est applicable car l'index et les R-Tree sont essentiellement composés de listes contiguës d'éléments simples et uniformes, sans pointeur ou indirection.

9 Benchmark

Les choix de structures de données et d'optimisations présentés au cours de cet article reposent sur un certain nombre d'hypothèses concernant les données à indexer. Ainsi, il est nécessaire de vérifier sur des données réelles si l'objectif est atteint à l'aide d'un benchmark. La trace d'exécution

utilisée pour ce benchmark a été obtenue avec le traceur Frida sur le programme *wget*, selon le protocole suivant :

```
# Ouvre un serveur HTTP qui sert un dossier contenant une large base de code
python -m http.server -b 127.0.0.1

# Lancement du traceur
python trace.py -t '*' spawn /bin/wget wget <MAIN_FN_ADDR> -a '/bin/wget,--recursive,--delete-after,http://localhost:8000'

# ... le traceur est arrêté lorsque la trace d'exécution contient 1 milliard d'instructions
```

Le benchmark a été réalisé à l'aide de la bibliothèque Rust "Criterion". Pour chaque type de requête et à chaque itération, des paramètres aléatoires sont générés, puis le temps d'exécution de la requête est mesuré. La figure 14 présente les résultats obtenus.

— La récupération des octets d'une plage d'adresses s'avère très rapide, avec une latence d'environ 1 µs. La taille de cette plage n'affecte que très légèrement le temps d'exécution.

— L'obtention de la valeur de chaque registre à un instant aléatoire requiert entre 300 ns et 100 µs. Certains registres étant beaucoup plus fréquemment mis à jour que d'autres (RIP, RAX, RSP, etc.), le temps de traitement de la requête est directement impacté par la quantité de rectangles présents dans leurs R-Tree respectifs. En moyenne, la récupération des valeurs des 17 registres à un instant aléatoire nécessite environ 2 ms.

— La variance la plus importante s'observe sur la requête cherchant les instants précédent et suivant où le registre PC vaut X, à partir d'un instant T. En effet, cette requête se distingue des autres : il s'agit d'une recherche d'intersection optimisant un paramètre spécifique, à savoir la distance par rapport à T. Pour la traiter efficacement, il a été nécessaire d'implémenter un algorithme de recherche distinct qui, lors du parcours, priorise les nœuds minimisant cette distance.

10 Conclusion

La conception de ce nouveau backend est l'aboutissement d'un an et demi de recherche et d'expérimentation, et les résultats sont très encourageants. La latence de la majorité des requêtes dépasse largement l'objectif initial, et la complexité de l'algorithme de construction de l'index est quasi linéaire par rapport à la taille de la trace d'exécution. De plus,

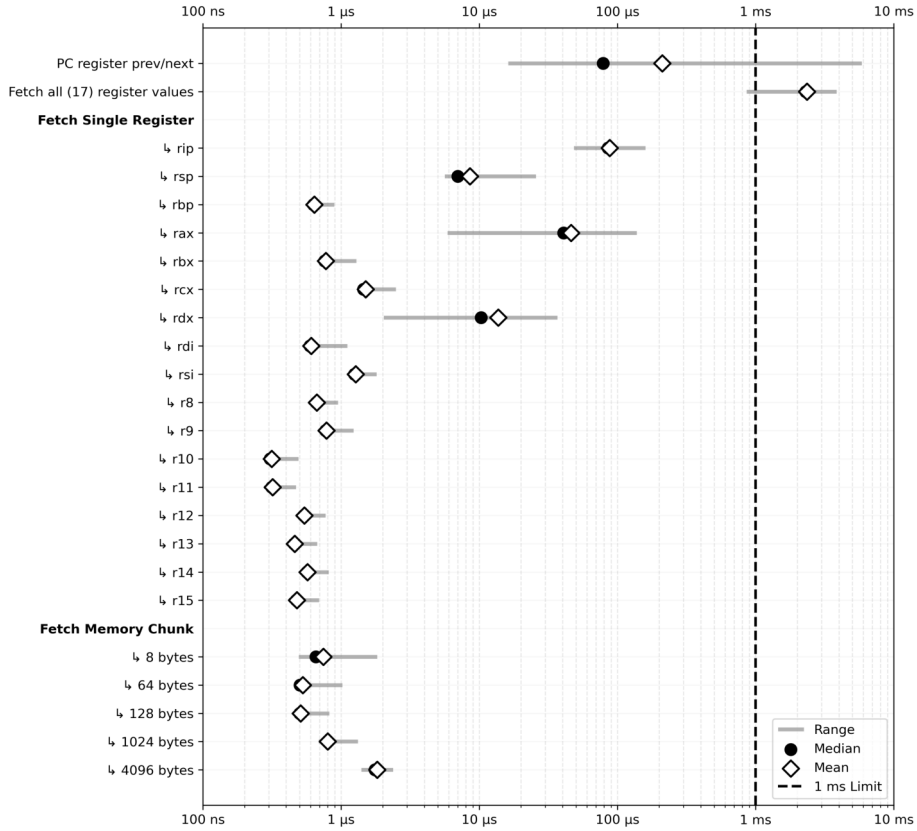

Fig. 14. Distribution de la latence par type de requête (échelle logarithmique)

la nouvelle architecture du projet permet d'exploiter l'index directement via une bibliothèque Rust afin de mener des analyses plus approfondies. Enfin, ce backend est si différent du précédent que nous avons décidé de redévelopper le frontend en partant de zéro. L'ensemble de ces composants sera rendu open-source lors du SSTIC 2026.

Références

1. Markus Gaasedelen. Tenet - a trace explorer for reverse engineers. `https://github.com/gaasedelen/tenet`, 2021.
2. I. Kamel and Christos Faloutsos. Hilbert r-tree : An improved r-tree using fractals. *Proc. Twentieth Int. Conf. Very Large Databases*, 10 1999.
3. Kyle Langendoen, Brad Glasbergen, and Khuzaima Daudjee. Nir-tree : A non-intersecting r-tree. In *Proceedings of the 33rd International Conference on Scientific*

and Statistical Database Management, SSDBM '21, pages 157–168, New York, NY, USA, 2021. Association for Computing Machinery.

4. Martin Perrier Louis Jacotot. Frinet - reverse-engineering made easier. `https://www.synacktiv.com/publications/frinet-reverse-engineering-made-easier`, 2023.
5. B. Moon, H.V. Jagadish, C. Faloutsos, and J.H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering*, 13(1) :124–141, 2001.