Cartographie des programmes et de leurs interrelations

Tristan Benoit

UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Cartographie des programmes et de leurs interrelations

THÈSE

présentée et soutenue publiquement le 13 décembre 2023

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Tristan Benoit

Composition du jury

Présidente :	Marine Minier	Université de Lorraine
Rapporteurs :	Christophe Hauser	Dartmouth College
	Davide Balzarotti	EURECOM Sophia Antipolis
Examinateurs :	Valérie Viet Triem Tong	CentraleSupélec
	Marine Minier	Université de Lorraine
	Yves Le Traon	Université du Luxembourg
Directeur de thèse :	Jean-Yves Marion	Université de Lorraine
Co-directeur de thèse :	Sébastien Bardin	CEA LIST

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thesul.

Remerciements

En premier lieu, je tiens à exprimer ma profonde gratitude à mes directeurs de thèse, Jean-Yves Marion et Sébastien Bardin. Merci à Jean-Yves, pour sa patience et son écoute, pour m'avoir permit de présenter mon travail à de nombreuses occasions et pour m'avoir octroyé une grande marge de manœuvre. Merci à Sébastien, pour son aide précieuse pour la rédaction et pour ses nombreuses heures passées à soumettre des articles scientifiques.

Je remercie vivement les membres du jury, Christophe Hauser et Davide Balzarotti les rapporteurs, et Valérie Viet Triem Tong, Marine Minier, Yves Le Traon les examinateurs pour leur engagement et l'intérêt manifeste pour mes travaux de recherche.

Je remercie évidemment les membres de l'équipe CARBONE. En particulier, Fabrice Sabatier pour m'avoir fournit de précieuses données et des machines à disposition et Guillaume Bonfante pour ses précieux conseils.

Je suis également reconnaissant envers l'école doctorale qui a offert des formations bien utiles, m'offrant l'occasion de rencontrer des collègues d'autres domaines.

Je tiens à remercier Marie Duflot-Kremer pour l'organisation de soirées jeux mensuelles mémorables ainsi que le service communication du LORIA, en particulier Maira Nassau et Mariana Diaz, pour leur aide.

L'équipe SEMAGRAMME mérite également mes remerciements pour leur hospitalité, en particulier Vincent pour ces soirées mémorables au bureau.

Il est impossible de saluer tous les collègues rencontrés dont la compagnie m'a permis de tenir, mais je remercie Grégoire pour son aide lors de la soumission de mes artéfacts et Athénaïs pour l'organisation des cafés doctorants.

Je souhaite remercier les professeurs qui m'ont marqué : Bruno Zanuttini et Florent Madelaine pour m'avoir transmis le goût de la recherche, Daniel Hirschkoff pour m'avoir accueilli en master ainsi que Laure Gonnord, sans qui les attendus du sujet de thèse, auraient été bien difficiles à intégrer.

Je tiens à remercier mes amis, Guillaume, Tristan, Antoine, Tanguy, David, Robin et Hoang de m'avoir donné la volonté de commencer cette thèse. Par ailleurs, je tiens à remercier Thomas pour les multiples rencontres lors des soirées qu'il a organisées, notamment avec Axel et Geoffrey.

Enfin, je ne pourrais pas terminer sans remercier du plus profond de mon cœur mes parents, pour m'avoir permis de réaliser mes études malgré les nombreuses difficultés. Merci pour votre soutien qui a toujours été présent.

Au logiciel FRoG Creator

Table des matières

Introduction

1 Contexte

La validité des logiciels est complexe à assurer. Certes, la disponibilité de bibliothèques par les gestionnaires de paquets et les dépôts en ligne ont permis l'essor de standard. Mais, l'utilisation de services de type question-réponse, tel que Stack Overflow [1], a intensifié la pratique de recopier des morceaux de codes, proposé par des utilisateurs, répondant à des problèmes spécifiques. Bien que le service implémente un système de vote afin de maintenir une bonne qualité dans les réponses, l'utilisation du service augmenterait le nombre de bugs [1]. En effet, même des réponses correctes peuvent être mal réutilisées si le contexte précis de la question est mal compris.

Pour assurer la qualité ou la sécurité des programmes, il est possible en principe de recourir aux méthodes formelles [9, 87, 138] avant la distribution des programmes [2]. Cependant, si le recours aux méthodes formelles peut permettre d'assurer la distribution d'un programme sûr ou efficace, il demande d'avoir une spécification précise à vérifier et n'est donc pas adapté à toutes les investigations possibles sur un programme – sans évoquer les problèmes potentiels de passage à l'échelle et d'expertise requise pour mener les analyses.

Nous nous intéressons de notre côté à des problèmes de rétro-ingénierie à haut niveau après la distribution. Là où la rétro-ingénierie à bas niveau cherche à comprendre la structure du programme [117] (p. ex., retrouver les instructions ou les fonctions du code), nous recherchons des informations sur un programme par rapport à son écosystème de production. Par exemple, savoir comment le programme a été créé (p. ex., déterminer sa chaîne de compilation), ou savoir si le programme est une variante d'un autre programme connu, ou s'il embarque des bibliothèques connues. Pour ce faire, nous n'utilisons pas des méthodes formelles, en effet ces tâches typiques de rétro-ingénierie se prêtent mal à la spécification formelle, mais de l'apprentissage automatique ou des comparaisons de différentes caractéristiques d'un programme.

Dans notre contexte, la seule information disponible est le programme final. En effet, afin d'être exécuté, le code source doit être traduit par la chaîne de compilation en un programme. Or, pour diverses raisons, il arrive qu'on n'ait pas accès au code source d'un programme. Par exemple, les entreprises ont intérêt à protéger leur propriété intellectuelle. Malheureusement, la traduction du programme au code source est d'une difficulté extrême [146].

1. `https://stackoverflow.com/`
2. Dans cette thèse, nous utilisons le terme de programme pour désigner des exécutables et des bibliothèques.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void)
{
    int entree, secret;
    secret = 42;

    printf("Quel est le nombre ?\n");
    scanf("%d", &entree);

    if (secret < entree)
    {
        return 0;
    }
    else if (entree > secret)
    {
        return 1;
    }
    return 2;
}
```

(a) Code source.

```
main:
        pushq   %rax
        movl    $.Lstr, %edi
        callq   puts
        leaq    4(%rsp), %rsi
        movl    $.L.str.1, %edi
        xorl    %eax, %eax
        callq   __isoc99_scanf
        xorl    %eax, %eax
        cmpl    $43, 4(%rsp)
        setl    %al
        addl    %eax, %eax
        popq    %rcx
        retq
.L.str.1:
        .asciz  "%d"

.Lstr:
        .asciz  "Quel est le nombre ?"
```

(b) x86-64 avec Clang 9.0.0 -O3.

```
main:
        push    {r11, lr}
        mov     r11, sp
        sub     sp, sp, #8
        ldr     r0, .LCPI0_0
        bl      puts
        ldr     r0, .LCPI0_1
        add     r1, sp, #4
        bl      scanf
        ldr     r1, [sp, #4]
        mov     r0, #0
        cmp     r1, #43
        movlt   r0, #1
        lsl     r0, r0, #1
        mov     sp, r11
        pop     {r11, lr}
        bx      lr
.LCPI0_0:
        .long   .Lstr
.LCPI0_1:
        .long   .L.str.1
.L.str.1:
        .asciz  "%d"

.Lstr:
        .asciz  "Quel est le nombre ?"
```

(c) armv7 avec Clang 9.0.0 -O3.

FIGURE 1 – Un code source compilé en 2 programmes par des chaînes de compilation différentes.

Nos travaux sont applicables à l'analyse de la chaîne d'approvisionnement d'un logiciel. L'identification de la chaîne de compilation et la détection des clones aident à repérer des vulnérabilités introduites par les compilateurs, ou des logiciels malveillants ainsi qu'à identifier les versions des bibliothèques. En cartographiant l'écosystème de production, nous pouvons reconnaître les composants à risque et les points d'entrée potentiels pour des attaques.

La Figure 1 présente les différences au niveau des instructions entre deux programmes compilés pour deux architectures matérielles différentes (x86 et ARMv7). Puisque ces deux programmes partagent le même code source, on dit que ce sont des *clones*.

Chaîne de compilation La chaîne de compilation [125] est l'ensemble des éléments servant à produire un programme à partir d'un code source. Elle est composée de la famille du compilateur (p. ex., Visual Studio ou Clang), de la version du compilateur (p. ex., 10.0 ou 12.0), des options du compilateur (p. ex., -O2 ou -Os), de l'addition d'offuscations intentionnelles, et de l'architecture cible du programme (p. ex., x86 ou ARM). Bien que plusieurs chaînes de compilations puissent être utilisées dans la compilation d'un unique programme, nous ne nous intéressons qu'au cas dans lequel une seule chaîne est utilisée pour toutes les parties du code source.

Recherche de clones L'étude des similarités de code binaire porte sur les similitudes et les différences entre deux ou plusieurs morceaux de code binaire [61]. Ces approches comparent le code binaire à différents niveaux de granularité (p. ex., blocs de base, fonctions, programmes). De notre côté, nous cherchons des clones de programmes entiers dans des dépôts. Un clone est un programme soit compilé à partir du même code que l'original avec une chaîne de compilation différente, soit à partir d'une autre version du même code source.

2 Problématique

Cette thèse a pour objet la rétro-ingénierie à haut niveau, et plus précisément l'identification d'un programme et la récupération d'informations concernant le processus de création d'un programme. Alors que la plupart des travaux récents considèrent la fonction binaire comme unité de travail [61], nous recherchons des informations sur les programmes entiers, le tout sans avoir accès au code source ni à des symboles laissés par le compilateur.

Plus précisément, sur un programme donné P, nous nous attaquons à deux tâches. La première tâche est de retrouver la chaîne de compilation utilisée pour produire P. La seconde tâche est de retrouver un clone de P dans une base de programme.

Tâche 1 : Prédiction de la chaîne de compilation Notre première tâche est de prédire la chaîne de compilation. Le flot d'un programme entier est largement influencé par les options de compilations, par exemple les extensions inline et des opérations de refactorisation telles que l'élimination de code inutile et la propagation de constantes. Par conséquent, une approche globale qui détermine une chaîne de compilation est appropriée pour des programmes compilés avec une seule chaîne de compilation. Cette connaissance sur les outils de compilation est d'importance capitale, car elle permet, d'une part, de détecter les vulnérabilités potentiellement injectées par ces outils [67], comme la vulnérabilité CVE-2018-12886 [3] produite par le compilateur GCC de la version 4.1 à 8. D'autre part, elle améliore l'identification des fonctions binaires [79], ce qui est crucial pour la maintenance et la sécurité des logiciels, la détection de clones et la rétro-ingénierie. Enfin, la connaissance de la chaîne de compilation peut aider dans les investigations numériques pour attribuer la création de programmes malveillants ou confirmer l'exploitation commerciale non autorisée de code source libre.

Tâche 2 : Recherche de clones de programmes Notre seconde tâche est de retrouver le clone d'un programme inconnu. Nous formulons cela sous le nom de *recherche de clones*. Le programme recherché est appelé *programme cible* et nous le recherchons dans un *dépôt*. Le dépôt contient au moins un clone ; notre objectif n'est ainsi pas de vérifier la présence d'un clone, mais plutôt de le retrouver. Ce problème est différent de l'équivalence fonctionnelle puisque les clones ne sont pas totalement sémantiquement équivalents. Prouver que des codes binaires ont les mêmes fonctionnalités est un problème théoriquement indécidable [94]. De plus, en pratique, cela ne peut être effectué que sur des morceaux de codes et pas avec des dépôts entiers de programmes [61]. Les clones peuvent avoir été compilés à partir du même code source avec une chaîne de compilation différente ou provenir d'une version légèrement différente du code source. Retrouver des clones donne une capacité très utile d'identification des programmes. D'un côté, détecter des logiciels malveillants [7, 41, 109, 152] est possible dès qu'un exemplaire du programme malveillant a été découvert. C'est crucial, car les jours suivant la découverte d'une vulnérabilité sont propices à des attaques puisque les vulnérabilités n'ont pas encore été corrigées [21]. Sans pouvoir corriger une

3. `https://nvd.nist.gov/vuln/detail/CVE-2018-12886`

vulnérabilité, la recherche de clones permet de détecter un programme malveillant et d'empêcher son exécution. De l'autre, l'identification de bibliothèques [43, 64, 140, 141] et la détection de plagiat [43, 64, 111] sont des applications immédiates de la recherche de clones.

3 Défis

Bien que le prisme des chaînes de compilation soit pertinent pour distinguer la sémantique (les fonctionnalités) d'un programme et son habillage, plusieurs défis doivent être considérés afin de produire des algorithmes efficaces et précis analysant des programmes entiers. Ces défis s'appliquent à nos deux tâches.

Défis 1 : Les chaînes de compilation sont extrêmement variées La plupart des éléments de la chaîne de compilation sont indépendants. Ainsi, tous les compilateurs ciblant les systèmes GNU Linux peuvent être compilés sur toutes les architectures sous-jacentes. De plus, chaque version des compilateurs est compatible avec ces architectures. Les compilateurs proposent plusieurs options indépendantes entre elles, par exemple des optimisations spécifiques portant sur les boucles, ou sur les appels des fonctions. Pour évaluer les résultats d'un outil sur chacune des deux tâches, il est nécessaire de disposer d'un jeu de données comportant des dizaines de chaînes de compilation.

Défis 2 : Les programmes sont des objets massifs Les programmes comptent des dizaines de milliers de fonctions binaires différentes qui s'appellent entre elles. Les fonctions contiennent elles-mêmes des suites d'instructions séquentielles appelées blocs de bases. Les blocs de bases sont liés entre eux par des sauts conditionnels. Intégrer des milliers d'instructions en un temps court est difficile. Pourtant, les caractéristiques extraites des programmes doivent être concises. Typiquement, les réseaux de neurones ne peuvent pas ingérer l'ensemble des instructions. L'analyse dynamique est bien trop longue sur des programmes importants. De plus, la vérification formelle risque d'avoir à explorer un nombre de chemins exponentiel en la taille du programme. Enfin, les motifs marquant une chaîne de compilation ou des fonctionnalités doivent être donc relevés globalement. Dans les deux tâches, le traitement du programme en entrée doit prendre en compte ces difficultés.

Défis 3 : Les bases de données de programmes sont larges Les programmes devant être compilés avec des dizaines de chaînes de compilation, les bases de données de programmes comptent des dizaines de milliers de programmes. Cela pose des difficultés dans les deux tâches. Pour prédire la chaîne de compilation, si l'on utilise de l'apprentissage, il est nécessaire d'utiliser des réseaux rapides pour pouvoir traiter toutes les données. La recherche d'un clone doit quant à elle être rapide. Soit la comparaison entre le programme cible et un candidat doit être rapide, soit une structure spécifique dans la base de données doit permettre de ne retenir qu'un petit nombre de candidats.

4 Contributions

Nous imaginons des techniques nouvelles afin de relever les défis présents dans nos deux tâches. Nos techniques ont pour points communs de cibler des programmes entiers et d'être rapides. La multiplicité des chaînes de compilations oblige à une certaine robustesse, en particulier en ce qui concerne les différentes architectures. Nous amenons trois contributions principales.

SNN Une solution pour prédire la chaîne de compilation, SNN (Chapitre 2) :

- Nous développons des réseaux de neurones sur les graphes, appelées *Site Neural Network* (SNN), pour déterminer la chaîne de compilation ayant généré un programme entier. SNN est capable de l'identifier au niveau du programme et obtient de meilleurs résultats qu'une méthode récente [105]. En particulier nos SNN sont 68 fois plus rapides durant l'apprentissage ;
- Nous évaluons SNN sur 120 chaînes de compilation différentes, dont plusieurs versions des compilateurs Clang, GCC, MinGW et Visual Studio, ainsi que plusieurs niveaux d'optimisations. Le jeu de données comprend environ 36 000 programmes.

PSS Une méthode de recherche de clones de programmes, PSS (Chapitres 3, 4 et 5) :

- Nous développons la méthode nommée *Program Spectral Similarity (PSS)* pour mesurer la similarité de programmes dans le cadre des recherches de clones de programmes sur de grands dépôts. Elle atteint un compromis idéal en matière de vitesse, de précision et de robustesse. PSS, et ses versions optimisées PSSO et PSSOH, sont robustes, même en cas de changement d'architecture et contre certaines offuscations. Par exemple, PSS trouve un clone dans 53% des cas face à l'obfuscation du bug de flot de contrôle. De plus, PSSOH prend seulement 0,39s pour une recherche sur un dépôt d'environ 85 000 programmes ;
- Nous mettons en place un cadre d'évaluation comprenant 15 méthodes provenant de l'état de l'art et couvrant les systèmes Linux et Windows ainsi que les objets connectés pour un total de plus de 200 000 programmes ;
- Nous portons des conclusions sur les différentes classes de méthodes possibles. Notamment le fait que de nombreux travaux antérieurs portant sur la recherche de clones de fonctions ne peuvent pas faire face à la recherche de clones de programmes en raison d'un manque de rapidité et de robustesse. Par exemple, Gemini [154] prend 2m en moyenne pour effectuer une recherche sur un petit jeu de données quand une version optimisée de PSS, PSSO, prend 0,27s. De plus, dans un scénario difficile, Gemini trouve un clone dans 29% des cas alors que PSSO trouve un clone dans 38% des cas.

Artéfacts Des artéfacts de nos recherches afin de promouvoir la science ouverte :

- Des artéfacts de SNN et de son cadre d'évaluation, comprenant le jeu de données d'environ 36 000 programmes complets ainsi que les codes sources utilisés et les expériences avec un autre outil ;

- Des artéfacts de PSS et de son cadre d'évaluation, comprenant 22 méthodes de recherches de clones dont une dizaine réimplémentées. Des résultats précalculés sont inclus à différents stades de la recherche de clone. En raison d'un manque d'espace de stockage, les plus de 200 000 programmes ne sont pas inclus. Cependant, les recherches de clones sont reproductibles.

5 Production scientifique

5.1 Publications

Le travail présenté dans les Chapitres 3 et 4 a été accepté à ESEC/FSE, une des toutes meilleures conférences en génie logiciel (rang A*) :

- [19] : *Scalable Program Clone Search Through Spectral Analysis*, Tristan Benoit, Jean-Yves Marion, Sébastien Bardin. Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2023.

Le travail du Chapitre 2 a été présenté à SANER, une conférence en rétro-ingénierie (rang A) :

- [20] : *Binary level toolchain provenance identification with graph neural networks*, Tristan Benoit, Jean-Yves Marion, Sébastien Bardin. Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2021.

5.2 Artéfacts

Comme déjà décrits, nous mettons à dispositions des artéfacts de nos recherches :

- SNN et son cadre d'évaluation sont disponibles sur le GitLab d'Inria [4] ;
- PSS et son cadre d'évaluation sont disponibles sur Zenodo [18].

5.3 Présentations

Workshops et Séminaires internationaux :

- Machine Learning for Program Analysis (MLPA) 2020. Workshop collocalisé avec IJCAI 2020 explorant les applications de l'apprentissage automatique à l'analyse des programmes ;
- 6th Franco-Japanese Cybersecurity Workshop, 2022. Groupe de travail franco-japonais explorant le domaine de la cybersécurité, organisé par Inria ;
- Sixième journée franco-allemande pour la cybersécurité, 2023. Journée de rencontre entre l'université de Lorraine et le CISPA (Helmholtz Center for Information Security) à Saarbrücken.

Séminaires nationaux :

- Réunion PEPR Cyber/DefMal 2023 au Campus Cyber. Rencontres nationales entre les acteurs du PEPR-Cybersécurité Defmal suivies de présentations scientifiques.

4. `https://gitlab.inria.fr/tbenoit/saner-2021-binary-level-toolchain-provenance-identification`

Présentation d'un poster :

- Rendez-Vous de la Recherche et de l'Enseignement de la Sécurité des Systèmes d'Information (RESSI) 2022. Rencontre portant sur la sécurité informatique organisée par le CNRS.

Séminaire interne :

- Groupe de recherche en informatique, image, automatique et instrumentation de Caen (GREYC), 2023. Séminaire Cryptologie & Sécurité.

6 Plan du document

Le manuscrit de thèse est organisé de la façon suivante :

- Le Chapitre 1 introduit le bagage scientifique nécessaire à la compréhension des Chapitres suivants en commençant par la compilation et la rétro-ingénierie avant d'aborder les éléments nécessaires à nos contributions tels que l'apprentissage automatique et la comparaison de graphes ;
- Le Chapitre 2 présente le problème de la prédiction de la chaîne de compilation et ses applications. Il décrit la méthodologie et notre modèle de réseau de neurones SNN ainsi que le jeu de données provenant de CodeForces. Des expériences avec un autre outil attestent de l'intérêt de notre méthode, notamment de sa rapidité. Enfin, une discussion et une conclusion mettent en perspective notre travail avec l'état de l'art ;
- Le Chapitre 3 aborde le problème de la recherche de clones de programme. Une définition du problème ainsi qu'un exemple posent les bases de notre paradigme. Notre contribution majeure PSS, une métrique de similarité de programme par l'analyse spectrale, est justifiée par des liens avec la distance d'édition de graphe. La complexité en temps théorique basse de notre méthode est mise en contraste avec l'état de l'art ;
- Le Chapitre 4 est une évaluation rigoureuse de PSS et de 18 autres méthodes de recherche de clones. Une évaluation portant sur des centaines de milliers de programmes démontre que PSS est un bon compromis entre la vitesse et la précision tout en étant très robuste, même en cas de changement d'architecture. De plus, quand des identificateurs littéraux tels que des chaînes littérales sont disponibles, une heuristique très simple est la méthode la plus précise ;
- Le Chapitre 5 étend le paradigme de la recherche de clones en intégrant des algorithmes de recherches rapides. Cela nous permet d'accélérer considérablement PSS, au prix d'une légère perte de précision de la recherche ;
- Enfin, la Conclusion résume les contributions puis ouvre des perspectives à la fois concernant l'apprentissage de vecteurs de programmes et l'analyse spectrale appliquée sur les programmes.

Chapitre 1

Bagage scientifique

Le Chapitre 1 pose les fondations scientifiques de cette thèse, en rappelant des concepts clés de l'informatique tels que la compilation de code source et les aspects essentiels de la rétro-ingénierie à bas niveau. Nous introduisons également les principales méthodes et outils d'apprentissage automatique, ainsi que le rôle crucial de la représentation sous forme de graphes pour comprendre et comparer des données. C'est à partir de ces bases que nous allons développer nos propositions innovantes pour la prédiction des chaînes de compilation (Chapitre 2) et la recherche de clones de programmes (Chapitres 3, 4 et 5).

1.1 Compilation

Les systèmes informatiques, du microprocesseur à l'architecture réseau, sont composés à la fois de composants matériels et de composants logiciels. Pour exécuter des commandes sur un ordinateur, il est préférable d'écrire un code dans un langage informatique, comme le langage C dans l'exemple du programme fibo en Listing 1.1 qui calcule la suite de Fibonacci . Ce code est souvent appelé code source. Il contient parfois des constructions de haut niveau telles que les classes ou de la métaprogrammation. Afin que l'ordinateur comprenne ce code source, il doit être converti d'une forme lisible en langage assembleur par un compilateur.

```
#include  <stdlib.h>
#include  <stdio.h>
int fibo(int n) {
    if (n <= 2) {return n ;}
    return fibo(n-1) + fibo(n-2) ;
}
int main(int argc, char *argv[]) {
    printf("%d", fibo( atoi(argv[1]) ) ) ;
    return 0 ;
}
```

Listing 1.1 – Code source en C du programme fibo implémentant la suite de Fibonacci.

```
fibo:   ; Fonction fibo et son premier bloc de base
        push    rbp  ; 55
        push    rbx  ; 53
        sub     rsp, 8  ; 48 83 ec 08
        mov     ebx, edi  ; 89 fb
        cmp     edi, 2  ; 83 ff 02
        jg      .L4 ; 7f 09
.L2:    ; Second bloc de base de fibo
        mov     eax, ebx ; 89 d8
        add     rsp, 8 ; 48 83 c4 08
        pop     rbx ; 5b
        pop     rbp ; 5d
        ret ; c3
.L4:    ; Troisieme bloc de base fibo
        lea     edi, [rdi-1]  ; 8d 7f ff
        call    fibo  ; e8 e2 ff ff ff
        mov     ebp, eax  ; 89 c5
        lea     edi, [rbx-2]  ; 8d 7b fe
        call    fibo  ; e8 d8 ff ff ff
        lea     ebx, [rbp+0+rax]  ; 8d 5c 05 00
        jmp     .L2  ; eb df
main:   ; Fonction porte d'entree du programme
        sub     rsp, 8  ; 48 83 ec 08
        mov     rdi, QWORD PTR [rsi+8]  ; 48 8b 7e 08
        mov     edx, 10  ; ba 0a 00 00 00
        mov     esi, 0  ; be 00 00 00 00
        call    strtol  ; e8 c5 fe ff ff
        mov     edi, eax  ; 89 c7
        call    fibo  ; e8 b4 ff ff ff
        mov     esi, eax  ; 89 c6
        mov     edi, OFFSET FLAT:.LC0  ; bf 00 00 00 00
        mov     eax, 0  ; b8 00 00 00 00
        call    printf  ; e8 9d fe ff ff
        mov     eax, 0  ; b8 00 00 00 00
        add     rsp, 8  ; 48 83 c4 08
        ret  ; c3
.LC0:   ; Chaine litterale utilisee pour afficher le resultat
        .string "%d"  ; 00 25 64 00
```

Listing 1.2 – Code assembleur du programme fibo pour l'architecture x86-64 créé par GCC.

Le code assembleur produit ensuite un code binaire qui peut ensuite être exécuté nativement par l'ordinateur. Ce code binaire est écrit dans un langage machine spécifique à l'architecture informatique. Il est composé d'instructions primitives utilisées pour contrôler le matériel, manipuler les données et interagir avec d'autres programmes. Par exemple, le Listing 1.2 contient un code assembleur pour l'architecture x86-64 de l'implémentation précédente de la suite de Fibonacci compilé avec GCC.

Suivant le système d'exploitation, différents formats de fichiers sont utilisés pour encoder des informations. Ainsi sur les systèmes Linux, le format ELF (Executable and Linkable Format) représente les programmes. Le format a pour début un entête de fichier de taille fixe précisant de nombreuses informations comme l'architecture cible, l'ordre de lecture des octets et le nombre de sections. Divers entêtes permettent de localiser et donc d'appeler des fonctions externes, c.a.d. des fonctions binaires présentes dans un autre programme. Enfin, certaines sections contiennent le code à exécuter tandis que d'autres contiennent des données. Cependant, rien ne garantit que la séparation entre le code binaire et les données soit respectée : on pourrait aisément exécuter ce qui est dans les données.

Rapport avec la thèse

La compilation est au cœur de cette thèse puisque le Chapitre 2 est centré autour de la prédiction de la chaîne de compilation d'un programme. De plus, les Chapitres 3, 4 et 5 sont focalisés sur la recherche d'un clone à un autre programme ayant été possiblement compilé avec une chaîne de compilation différente.

1.2 Rétro-ingénierie à bas niveau

La rétro-ingénierie à bas niveau consiste à partir de la suite des octets à obtenir les différents chemins possibles de l'exécution de ce code binaire. Le processus de décryptage du code binaire se fait généralement en plusieurs étapes. Il faut d'abord séparer les données des instructions durant le désassemblage pour obtenir les blocs de bases. Ensuite, il est nécessaire de retracer le graphe de flot de contrôle (CFG) et de le réécrire pour le simplifier. Puis, on localise les différentes fonctions du programme. Enfin, on peut créer le graphe d'appels de fonctions. La connaissance de toutes ces structures permet de comprendre le programme. À titre d'exemple, le code hexadécimal correspondant au programme fibo en Listing 1.2 est `55 53 48 83 ec 08 89 fb 83 ff 02 7f 09 89 d8 48 83 c4 08 5b 5d c3 8d 7f ff e8 e2 ff ff ff 89 c5 8d 7b fe e8 d8 ff ff ff 8d 5c 05 00 eb df 48 83 ec 08 48 8b 7e 08 ba 0a 00 00 00 be 00 00 00 00 e8 c5 fe ff ff 89 c7 e8 b4 ff ff ff 89 c6 bf 00 00 00 00 b8 00 00 00 00 e8 9d fe ff ff b8 00 00 00 00 48 83 c4 08 c3`.

Désassemblage Découper et décoder ce code en plusieurs séquences d'instructions est l'étape la plus importante de la compréhension du code binaire. C'est ce en quoi consiste le désassemblage [117]. Les désassembleurs tentent de séparer les données et les instructions. Le désas-

semblage statique [133] essaye de suivre les déplacements du pointeur de l'instruction actuelle. Cependant, étant donné que le code est composé d'une séquence d'octets et que le pointeur de l'instruction actuelle peut être déplacé n'importe où par l'instruction de saut `jmp`, il est très difficile de suivre tous les chemins possibles. Pour pallier ce problème, l'analyse dynamique simule d'exécution du programme afin d'obtenir les valeurs concrètes du pointeur d'instruction [22]. Néanmoins, la couverture de l'analyse dynamique est incomplète. Premièrement, les programmes dépendent de leur environnement, et si celui-ci ne répond pas correctement (p. ex., une communication réseau est absente) alors la reproduction de l'exécution est fausse. Deuxièmement, il est difficile de déterminer les conditions d'arrêts d'une boucle et l'exploration peut manquer certains chemins [28].

Graphe de flot de contrôle À partir du désassemblage, on peut construire le *graphe de flot de contrôle (CFG)* du programme entier. Le CFG est un graphe dirigé dont les chemins sont les différents chemins d'exécution possibles. Il aide à visualiser la structure du programme et à identifier les boucles et les branches conditionnelles. Les sommets d'un CFG sont appelés les *blocs de bases*, et ce sont des suites d'instructions séquentielles terminant éventuellement par un saut. Un des problèmes rencontrés lors de la construction du CFG à partir des blocs de bases est le chevauchement durant l'exécution. La même portion de code peut être rencontrée plusieurs fois durant l'exécution, mais ne pas avoir le même point d'entrée, ce qui crée un bloc de base différent pour chaque point d'entrée rencontrée. De plus, les boucles imbriquées sont difficiles à comprendre puisque la fin d'une des boucles est le début d'une autre, ce qui signifie que les deux boucles partagent les mêmes blocs de bases. Pour gérer ces problèmes, il faut réécrire le graphe de flot de contrôle [10,13,110]. De plus, le désassemblage statique a tendance à fortement surestimer les chemins possibles, il faut alors utiliser des analyses comme l'interprétation abstraite quitte à manquer des chemins potentiels [86]. De toute façon, même les programmes bien désassemblés ont des milliers de blocs de bases c'est pourquoi comprendre un programme avec son CFG complet est difficile et on découpe les programmes en fonctions.

Découpage en fonctions Une *fonction* binaire est un morceau de code avec un point d'entrée et un point de sortie bien définis. Elle est appelée par l'instruction de saut `call`, puis au moment de l'instruction `ret` le pointeur de l'instruction actuelle revient là où il était avant l'appel. Ce sont de petites unités comportant une dizaine de blocs de bases. Les fonctions n'ont qu'un point d'entrée, mais plusieurs points de sorties. Lorsque l'instruction `call` est utilisée, on sait qu'une fonction commence à cet endroit (pourvu que la position de la fonction soit identifiable par l'argument de l'instruction). Néanmoins, on ne sait pas directement quel est le périmètre de cette fonction. On doit donc partir de son début et marquer comme faisant partie de la fonction tous les blocs de bases rencontrés jusqu'aux points de sorties [3].

Fonctions de l'exemple Dans l'exemple très simple du programme fibo en Listing 1.2, il est facile de découper le programme en deux fonctions. La première est la porte d'entrée du

programme, qui procède à la conversion en entier d'une entrée textuelle (par les fonctions `atoi` et `strtol` en C), puis à un saut vers la fonction `fibo` et enfin à l'affichage du résultat par la fonction `printf`. La seconde est la fonction fibo, qui contient trois blocs de bases. Le premier bloc de base effectue la condition `if (n <= 2)` par l'enchaînement des instructions `cmp` et `jg`. En effet, l'instruction `cmp edi, 2` effectue la comparaison entre l'indice et le nombre 2 et enregistre le résultat dans un drapeau. De son côté, `jg` déplace le pointeur de l'instruction actuelle si le résultat de la comparaison est positif. Le second bloc de base est exploré quand l'indice est plus petit que 2, c'est alors que le résultat est renvoyé. Le troisième bloc de base est exploré quand l'indice est plus grand que 2, c'est alors que deux appels récursifs à la fonction `fibo` sont réalisés. Le résultat est renvoyé en passant par le second bloc.

CFG d'une fonction Pour analyser la structure d'une fonction, on construit un CFG qui lui est propre. Par exemple, le CFG de la fonction `fibo` en Figure 1.1 n'a pas de cycle, et donc la fonction n'a pas de boucle. En fait, c'est la récursion, par les instructions `call fibo`, qui permet de calculer le résultat. Les problèmes liés à la construction du CFG d'une fonction sont les mêmes que ceux liés à la construction du CFG complet du programme entier. À la différence que les fonctions n'ont qu'un nombre faible, en général inférieur à 20, de blocs de bases. Cela rend les réécritures plus simples de même que l'analyse de la fonction.

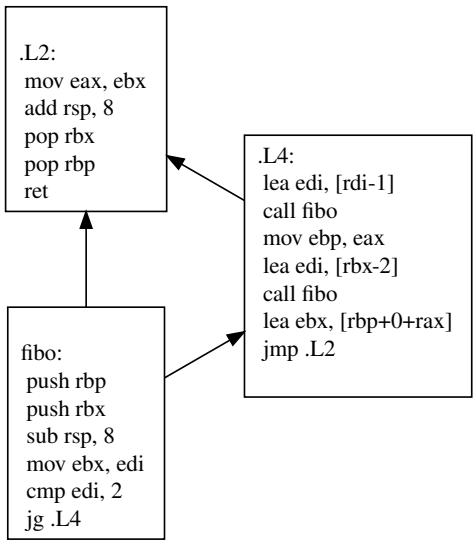


FIGURE 1.1 – Le graphe de flot de contrôle de la fonction binaire `fibo`.

Graph d'appels de fonctions Pour analyser les appels entre les fonctions, on utilise le graphe d'appels de fonctions. Ce graphe dirigé connecte une fonction lorsqu'elle appelle une autre fonction. Il est facile à obtenir après le découpage des fonctions puisque le découpage a besoin de comprendre les appels de fonctions. Dans le graphe d'appels de fonctions du programme fibo, en Figure 1.2, la récursivité est apparente par la boucle sur la fonction `fibo`.

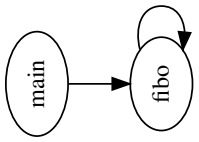


FIGURE 1.2 – Le graphe d'appels de fonctions du programme fibo.

Outils standard Pour analyser le code binaire, il existe une variété d'outils aux approches différentes, mais dont les résultats sont corrects de notre point de vue. Puisque nous nous concentrerons dans cette thèse sur la structure générale de programmes, la question de l'analyse à bas niveau des codes binaires n'est pas le sujet de cette thèse. Les analyses de code binaire peuvent être statiques, dynamiques ou symboliques. L'analyse statique [151] se contente de traduire les instructions en suivant le pointeur du mieux possible afin de produire les représentations du programme sous forme de graphes que nous venons de voir. Parmi les outils d'analyse statique, on trouve IDA [57], un logiciel payant très populaire grâce à sa gestion de nombreuses architectures matérielles et son automatisation par des greffons. Tout comme Binary Ninja [73], il offre une bonne interaction avec son interface puisqu'on peut modifier le programme durant l'analyse. Ghidra [47] est une alternative libre et gratuite développée par l'agence de sécurité des États-Unis (NSA). Enfin, Radare2 [142] est un outil en ligne de commande qui est basé sur Ghidra. L'analyse dynamique [48] profite en plus de l'exécution du programme pour parfaire l'exploration. QEMU [16] est un émulateur libre qui gère plusieurs architectures matérielles. Pintool [102] quant à lui est un outil d'Intel pour inspecter et manipuler l'exécution d'un programme en temps réel, ce qui apporte une granularité supplémentaire à l'analyse. L'analyse dynamique symbolique utilise des représentations formelles pour trouver les entrées qui provoqueront l'exécution d'une nouvelle partie du programme. Les logiciels BinSec [36, 37] et BOA [28] sont deux exemples d'analyse dynamique symbolique. Le fuzzing [98] est une autre technique dynamique qui explore les chemins par des entrées aléatoires.

Rapport avec la thèse

Dans le cadre de cette thèse, la rétro-ingénierie à bas niveau, bien qu'importante, n'est pas notre sujet d'étude, mais plutôt un préalable nécessaire pour fournir les représentations sous forme de graphes que nous utilisons. Ainsi, bien que la rétro-ingénierie à cette échelle ne soit pas notre sujet d'analyse primaire, il apporte les fondations nécessaires pour construire et déployer nos techniques de recherche. L'élaboration des graphes de flot de contrôle et la construction du graphe d'appels de fonctions sont essentielles à nos propositions tout au long de cette thèse. Plus précisément, le graphe d'appels est fondamental à notre proposition pour rechercher des clones PSS dans les Chapitres 3, 4 et 5, tandis que les graphes de flot de contrôle sont indispensables tout au long des différents Chapitres, et donc non seulement pour l'approche PSS, mais aussi pour nos réseaux de neurones basés sur les sites (SNN). Ces deux tâches peuvent être considérées comme de la rétro-ingénierie à haut niveau.

1.3 Apprentissage automatique

L'apprentissage automatique [128] est un champ d'études de l'intelligence artificielle qui se fonde sur des méthodes mathématiques, notamment d'optimisation, et de statistique pour apprendre à partir de données. L'apprentissage automatique comporte généralement deux phases. La première phase dite d'apprentissage ou d'entraînement consiste à entraîner un modèle à partir d'un nombre fini d'instances. Le modèle doit résoudre une tâche pratique, telle que prédire si une image contient un chat, le mot suivant dans une phrase ou la quantité d'argent dont dispose un client potentiel. La seconde phase dite d'évaluation cherche à évaluer le modèle. De nouvelles instances, auxquelles le modèle n'avait pas accès, sont soumises au modèle. Afin d'évaluer la qualité des prédictions, il est impératif de connaître les étiquettes liées aux instances, c.a.d. les réponses que devrait fournir le modèle.

Si les étiquettes sont discrètes, on parle d'un problème de *classification* sinon d'un problème de régression. Par exemple, lorsqu'on cherche à savoir si deux fonctions binaires sont similaires, il y a deux catégories : similaire ou non similaire. Les étiquettes sont discrètes, c'est donc un problème de classification. L'apprentissage automatique s'applique à différents types d'instances, tels que des graphes, des matrices ou des vecteurs, contenant des variables continues ou discrètes.

Apprentissage supervisé Si les classes sont prédéterminées et les étiquettes connues, on parle alors d'apprentissage supervisé [62]. L'optimisation est au cœur des méthodes d'apprentissage supervisées. Notamment, la descente de gradient [4] permet d'affiner des paramètres de fonctions dérivables en allant dans une direction afin de minimiser une fonction objectif. La fonction objectif évalue la perte entre la prédiction actuelle du modèle et l'étiquette connue. Suivant la nature des étiquettes et la distribution des étiquettes, plusieurs fonctions objectifs sont adaptées. Par exemple, dans le cas de la prédiction de la similarité entre des fonctions, la très large majorité des fonctions ne sont pas similaires entre elles. Il devient alors nécessaire d'utiliser une fonction objectif spéciale et de modifier l'apprentissage comme proposé par Gemini [154] ou αDiff [101].

Apprentissage non supervisé Quand la méthode n'utilise pas les étiquettes durant l'entraînement, on parle d'apprentissage non supervisé [14]. L'algorithme induit de lui-même la structure sous-jacente des entrées. Pour ce faire, le partitionnement des sous-ensembles de points proches est possible. Des outils mathématiques tels que la factorisation de matrice permettent de déceler les composantes principales des données et donc de réduire la dimension. L'apprentissage non supervisé permet d'analyser les données sans travail d'étiquetage, et il peut suffire à remarquer différentes chaînes de compilations (voir l'analyse en Section 2.6.2).

Apprentissage semi-supervisé L'apprentissage semi-supervisé [30] combine l'apprentissage sur des données étiquetées et non étiquetées. Ces méthodes sont utiles lorsqu'on souhaite classifier des données dans des catégories bien définies, mais qu'on ne dispose que peu de données étiquetées [131]. De plus, les étiquettes doivent être correctes ; or ce n'est pas toujours le cas. Par

exemple, les antivirus ne sont pas fiables sur de nouveaux programmes malveillants. Il peut être intéressant de n'utiliser que les étiquettes de meilleure qualité pour l'apprentissage.

Apprentissage auto-supervisé L'apprentissage auto-supervisé [30] construit un problème d'apprentissage supervisé depuis un problème non supervisé à l'origine. Sur une instance non étiquetée, on apprend à réaliser une sous-tâche dont l'instance comme l'étiquette sont des parties de l'instance d'origine. Une instance non étiquetée offre plusieurs instances étiquetées.

Auto-supervision pour vectoriser les fonctions binaires Dans le cas du langage, l'apprentissage auto-supervisé essaye de retrouver un mot en particulier suivant le contexte. Puisque le document complet a un effet sur la prédiction, un vecteur représentant le document complet est appris. Ce produit de l'apprentissage peut être alors considéré comme un vecteur caractéristique de l'instance non étiquetée. Par exemple, la méthode de vectorisation de fonctions binaires Asm2Vec [42] apprend à prédire une instruction grâce aux instructions autour. Des vecteurs communs à toutes les fonctions binaires sont associés aux instructions pour apprendre des vecteurs d'instructions. Un vecteur est associé à chaque fonction binaire et est utilisé par l'apprentissage pour discerner ce qui est spécifique à la fonction. Après l'apprentissage, un vecteur représente la fonction associée. De plus, la distance entre deux vecteurs est une mesure de la similarité des fonctions sous-jacentes.

1.3.1 Évaluation des modèles appris

Dans les cas où on ne dispose pas de données étiquetées pour la phase d'évaluation, des métriques d'évaluations spécifiques existent suivant la technique utilisée. Par exemple, les partitions [153] peuvent être évaluées par le coefficient de silhouette [127].

Proportion de succès Quand les étiquettes sont présentes, il est facile d'évaluer la capacité de prédiction des modèles. Dans les cas les plus simples, il est possible de calculer la proportion de bonnes prédictions. Néanmoins, la proportion de bonnes prédictions est peu utile lorsque certaines classes sont bien plus présentes que d'autres. Par exemple, s'il n'y a qu'un pour cent de programmes malveillants rencontrés, un classifieur qui renvoie toujours que le programme n'est pas malveillant aura une proportion de succès de 99%. C'est pourquoi nous utilisons une *matrice de confusion*, et nos deux mesures de base pour évaluer les prédictions d'un classifieur sont la *précision* et le *rappel*. La précision du système décrit précédemment sera toujours proche d'être parfaite, mais le rappel des programmes malveillants sera de 0 puisqu'aucun n'est détecté.

Matrice de confusion La matrice de confusion permet de visualiser simplement la qualité d'un modèle de classification. Chaque ligne correspond à une *classe réelle*, chaque colonne correspond à une *classe attribuée*. Une cellule contient le nombre d'instances d'une classe qui ont été attribuées comme appartenant à une certaine classe. Le total d'une ligne donne le nombre d'attributions

totales d'une classe. Le total d'une colonne donne le nombre d'instances appartenant réellement à une classe. Un exemple de matrice de confusion est donné par la Table 1.1.

Précision, rappel et F1-Score La précision pour une classe est définie comme le nombre d'instances correctement attribuées à cette classe divisé par le nombre total d'instances attribuées à cette classe. Intuitivement, la précision mesure la capacité à ne pas surévaluer le nombre d'instances d'une classe. Le rappel pour une classe est le nombre d'instances correctement attribuées à cette classe divisé par le nombre d'instances réelles de cette classe. On l'appelle aussi la sensibilité d'une classe, car cela mesure la capacité à ne pas sous-évaluer le nombre d'instances d'une classe. Pour combiner la précision et le rappel, on calcule le F1-Score : $\frac{2 \times Precision \times Rappel}{Precision + Rappel}$. L'usage d'une multiplication dans le numérateur oblige à garantir simultanément une bonne précision et un bon rappel si on veut obtenir un bon F1-Score.

Exemple Dans la matrice ci-dessous, la précision pour le chat est de $34/(34 + 16 + 1) = 34/51 = 2/3$. Le rappel pour le chat est de $34/(34 + 12 + 3) = 34/49$. Le F1-Score pour le chat est de $2 \times \frac{(2/3)\times(34/49)}{(2/3)+(34/49)} = 17/25$. Pour le chien, la précision est de $21/36$, le rappel de $21/41$, et le F1-Score de $6/11$. Pour la tortue, la précision est de $10/17$, le rappel de $10/14$, et le F1-Score de $20/31$. Le F1-Score moyen de toutes les classes est de $\frac{17/25+6/11+20/31}{3} = 15947/25575 \approx 0,623$.

Table 1.1 – Exemple de matrice de confusion sur 3 classes.

	Chat	Chien	Tortue	Total
Chat	34	16	1	51
Chien	12	21	3	36
Tortue	3	4	10	17
Total	49	41	14	104

Rapport avec la thèse

L'apprentissage automatique est une approche très utilisée durant cette thèse. Premièrement, notre proposition SNN dans le Chapitre 2 est une méthode d'apprentissage supervisé sur les graphes afin de prédire la chaîne de compilation d'un programme. Nous évaluons d'ailleurs les performances de celle-ci avec des statistiques comme la précision et le rappel. Deuxièmement, si notre proposition PSS n'utilise pas d'apprentissage automatique dans les Chapitres 3, 4 et 5, ce n'est pas le cas des autres méthodes auxquelles nous comparons PSS. Ainsi, dans les Chapitres 3 et 4, nous comparons PSS avec plusieurs méthodes basées sur l'apprentissage automatique. Par exemple, des méthodes de vectorisation de fonctions binaires, comme Asm2Vec [42], Gemini [154], SAFE [106] et αDiff [101]. Comme décrit dans cette Section, Asm2Vec a recours à l'apprentissage auto-supervisé, en revanche les autres méthodes sont supervisées. Nous étudions également DeepBinDiff [44] qui a recours à de l'apprentissage auto-supervisé pour comparer des programmes.

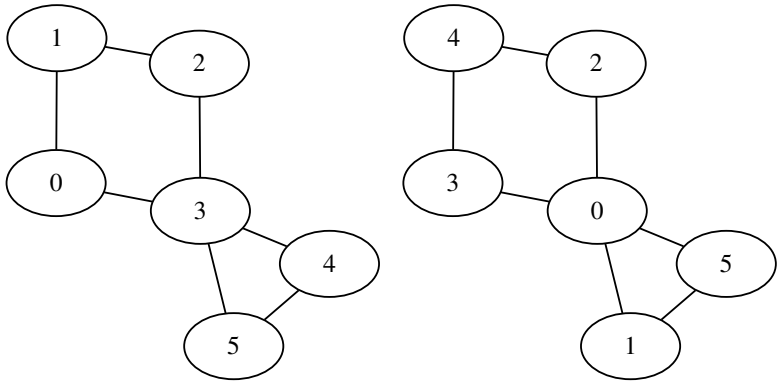
1.4 Comparaison de graphes

Les graphes sont des structures mathématiques utilisées pour représenter des relations entre des objets. Ils sont composés de sommets, également appelés nœuds, reliés entre eux par des arêtes. Les graphes fournissent une manière intuitive de visualiser et d'analyser des liens entre différents éléments.

Dans un graphe non dirigé, les arêtes ne possèdent pas de direction. Cela signifie qu'il est possible de se déplacer dans les deux sens le long de chaque arête. En revanche, dans un graphe dirigé, chaque arête possède une direction spécifique, ce qui signifie qu'il n'est pas possible de se déplacer dans les deux sens le long de chaque arête.

Plus formellement, un graphe G est muni d'un ensemble V de sommets et d'un ensemble E d'arêtes. Pour représenter un graphe, on utilise une matrice d'adjacence A de dimension $|V| \times |V|$, où $a_{i,j}$ est égal à 1 si $(V_i, V_j) \in E$ et à 0 sinon.

Le graphe G_1 en Figure 1.3a a pour matrice d'adjacence A_1.



(a) Un graphe G_1. (b) G_2 isomorphe à G_2.

$$A_1 = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \quad A_2 = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

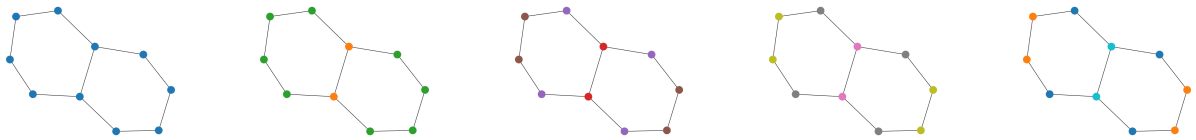
Isomorphisme de graphe On peut permuter les indices des sommets de G_1 afin d'obtenir un autre graphe G_2 dont la matrice d'adjacence est A_2. Le problème de l'isomorphisme de graphe est simplement de savoir si deux graphes sont les mêmes modulo une permutation des sommets, on dit alors qu'ils sont isomorphes. Ce problème n'est pas résoluble en temps polynomial [88], ce qui donne une idée d'à quel point comparer et analyser des graphes est une tâche ardue.

Test de Weisfeiler-Lehman Le test de Weisfeiler-Lehman (Algorithme 1) permet de distinguer une partie des graphes. L'algorithme assigne une couleur à chaque sommet. Puis, à chaque étape, une nouvelle couleur par sommet est assignée en concaténant les couleurs du voisinage ouvert de celui-ci. La sortie consiste en une matrice contenant pour chaque étape les couleurs de chaque sommet. Les couleurs peuvent être comparées indépendamment de l'ordre des sommets, car on peut trier les couleurs à chaque étape. Cependant, certains graphes non isomorphes ne peuvent jamais être distingués par cette méthode. Par exemple, le graphe en Figure 1.4 a toujours les mêmes couleurs à chaque étape que le graphe en Figure 1.5.

Algorithm 1 Test de Weisfeiler-Lehman.

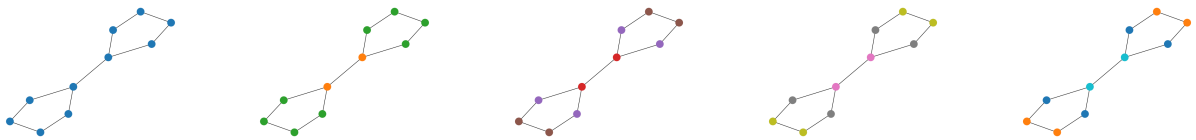
Entrée : $G = (V, E)$, le graphe.
Paramètre : l, le nombre d'étapes.
Sortie : Z, l'empreinte du graphe et une matrice de dimension $l \times |V|$.

Pour tout $u \in V$ **Faire**
  $Z_{0,u} \leftarrow 1$
Fin Pour
Pour $1 \leq i \leq l$ **Faire**
  Pour tout $u \in V$ **Faire**
    $Z_{i,u} \leftarrow hash(\{Z_{i-1,u}\} \cup \{Z_{i-1,v} | (u, v) \in E\})$
  Fin Pour
Fin Pour
Renvoyer Z



(a) (b) (c) (d) (e)

FIGURE 1.4 – Application du test de Wesfeiler-Lehman en 5 étapes.



(a) (b) (c) (d) (e)

FIGURE 1.5 – Application du test de Wesfeiler-Lehman en 5 étapes sur un autre graphe.

Cas des graphes dirigés Nous devons tout de même préciser que les graphes d'appels et les graphes de flot de contrôle sont des graphes dirigés qui ont pour racine le début du programme. Or, le problème de l'isomorphisme sur de tels graphes peut être résolu en temps polynomial [83].

Comparaison sans équivalence Toutefois, dans cette thèse, nous ne cherchons pas à résoudre le problème de l'isomorphisme de graphe. Notre premier problème dans le Chapitre 2 est de rechercher la chaîne de compilation qui a produit un graphe de flot de contrôle. Or, deux CFG peuvent être très différents, mais posséder la même chaîne de compilation. Notre second problème dans les Chapitres 3, 4 et 5 demande de pouvoir mesurer la similarité entre des graphes dirigés. Ces graphes ne sont pas nécessairement isomorphes, et nous ne recherchons pas une équivalence stricte entre un programme et son clone. En effet, d'une part cela serait trop coûteux à calculer [61] et d'autre part les clones peuvent avoir des versions légèrement différentes d'un même code source.

Distance d'édition de graphe La distance d'édition de graphe (GED) [56] est une façon précise de mesurer la similarité entre des graphes. La distance d'édition est le coût du plus petit chemin d'édition entre deux graphes. Un chemin d'édition P entre des graphes G_1 et G_2 est une séquence d'opérations $e_1, e_2, ..., e_n$. Les opérations d'édition de graphe incluent généralement la suppression ou l'ajout d'un sommet ou d'une arête. Chaque opération e_i est associée à un coût $c(e_i)$. Par exemple, le coût de la suppression d'un sommet peut être égal à son degré. Le principal inconvénient du calcul de GED est que c'est un problème NP-complet [160]. Même les approximations ont un temps $O(n^3)$ [134] où n est le nombre de sommets des graphes.

Rapport avec la thèse

L'étude des graphes est l'outil principal proposé dans cette thèse. Premièrement, notre proposition SNN dans le Chapitre 2 utilise des réseaux convolutifs sur les graphes de flot de contrôle. Ces réseaux se sont inspirés du test de Wesfeiler-Lehman afin de produire des représentations sous forme de vecteur des graphes. Deuxièmement, notre proposition PSS, dans les Chapitres 3, 4 et 5, fournit une ébauche de distance d'édition des graphes par une distance spectrale qui est rapide à calculer. Enfin, nous comparons PSS avec plusieurs méthodes calculant la similarité de graphes d'appels par des distances d'édition [54, 69, 129].

Chapitre 2

Prédiction de la chaîne de compilation

Le Chapitre 2 se focalise sur le problème de prédiction de la chaîne de compilation à partir d'un programme entier. Nous dressons un panorama détaillé du contexte, de la problématique ainsi que de l'état de l'art sur ce sujet. La proposition majeure de ce chapitre est notre modèle de réseau neuronal innovant Site Neural Network (SNN), spécialement conçu pour être utilisé dans des hiérarchies de classifieurs afin d'identifier de nombreuses variétés de chaînes de compilation. Nous menons des expériences avec un jeu de données conséquent, composé de plus de 36 000 programmes et comprenant 120 chaînes de compilation différentes. Les résultats de SNN par rapport à une méthode récente sur la prédiction de la chaîne de compilation sont mis en avant, soulignant la précision et la grande rapidité des hiérarchies de SNN. Notre approche est notamment 68 fois plus rapide que son concurrent durant l'apprentissage. Une brève discussion et une conclusion permettent d'élargir les perspectives de notre travail vis-à-vis de l'état de l'art.

2.1 Introduction

La *chaîne de compilation* ayant servi à produire un programme est composée de la famille du compilateur (p. ex., Visual Studio ou GCC), la version du compilateur (p. ex., 10.0 ou 12.0) ainsi que le niveau d'optimisation (p. ex. ou -O1, -O2). L'architecture ou la présence d'une offuscation font également partie de la chaîne de compilation, mais ne seront pas explorées dans ce premier travail.

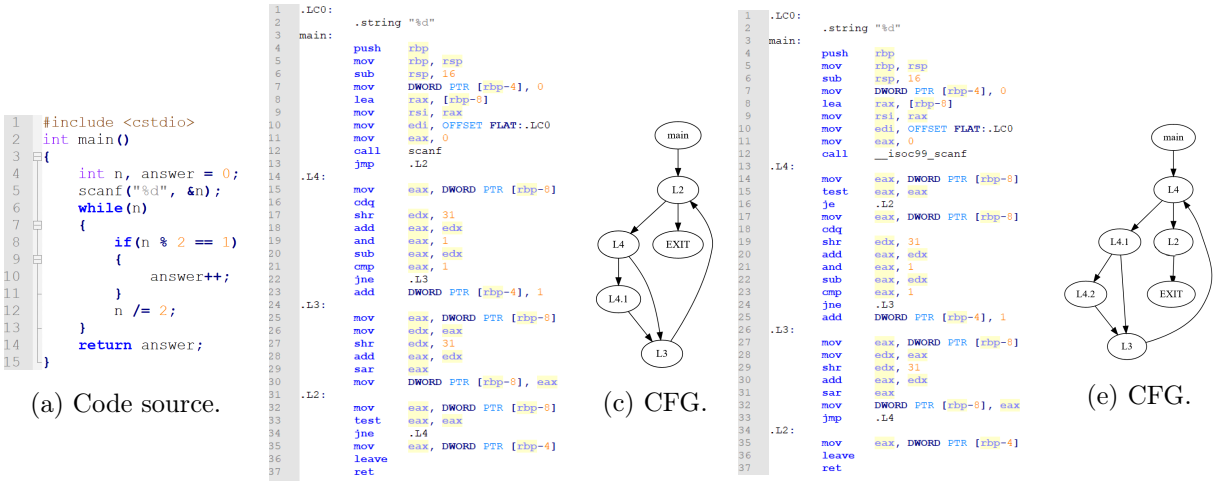
Les applications sont souvent construites par l'assemblage de produits informatiques standard (COTS), dans le cadre d'une chaîne d'approvisionnement logicielle qui comprend aussi des composants, des bibliothèques, des outils et des processus utilisés pour développer, construire et publier un logiciel. L'utilisation de COTS et de bibliothèques libres permet en principe une conception simplifiée et plus rapide, de plus ce sont des produits éprouvés donc fiables. Cependant, cette approche a certaines limitations. L'intégration de différents composants peut requérir un effort significatif. Il y a aussi une dépendance envers le mainteneur ou le fournisseur du produit. Des problèmes de sécurité peuvent également émerger, ainsi que des incompatibilités entre les composantes logicielles.

Bien que le code source, les symboles de débogage et la chaîne de compilation soient généralement disponibles dans le cas des bibliothèques, il n'est pas rare que ces informations soient perdues avec le temps. Pour ce qui est des COTS, leur code source est le plus souvent inaccessible, et le logiciel est habituellement dépouillé de ses symboles de débogage pour protéger la propriété intellectuelle du fournisseur. La chaîne de compilation des COTS est également inconnue. Déterminer la chaîne de compilation d'un programme dépouillé de ses symboles est un problème important dans au moins trois scénarios.

Détection de failles de sécurité. La question de la chaîne de compilation est importante pour la maintenance des logiciels et le support à long terme, car les compilateurs peuvent injecter des vulnérabilités découvertes bien après la sortie des COTS et après le déploiement d'applications les incorporant [67]. Par exemple, GCC de la version 4.1 à 8 révèle parfois l'adresse de la donnée contrôlant l'intégrité de la pile, introduisant ainsi la vulnérabilité CVE-2018-12886 qui permet l'attaque par débordement de la pile. De plus, tout un éventail de vulnérabilité est produit par les choix spécifiques des compilateurs face à des comportements dits indéfinis en langage C. Certains choix arbitraires engendrent des vulnérabilités telles que CVE-2016-9843 et CVE-2016-9840. Il est donc nécessaire de pouvoir retrouver la chaîne de compilation pour évaluer si une application peut contenir une faille de sécurité. De plus, l'ajout de la chaîne de compilation dans les rapports d'échecs facilite les diagnostics concernant les bugs liés aux implémentations des compilateurs, aux options ou à des incompatibilités entre les logiciels et les périphériques [115].

Identification de fonctions connues. L'identification des fonctions dans un code binaire est primordiale pour la maintenance et la sécurité des logiciels. Cela concerne entre autres la détection des clones [149] et la rétro-ingénierie des logiciels malveillants [26]. Cette identification ne pose pas de difficultés lorsque le code binaire contient suffisamment d'informations pour le désassembler. Par exemple, IDA propose l'algorithme FLIRT [57] basé sur la reconnaissance de motif, tandis que ByteWeight [12] construit un arbre de préfixe d'après les prologues de fonctions connues. Ces méthodes échouent dans de nombreuses situations : bibliothèques inconnues, légère modification du code, dépouillage des symboles ou offuscation. L'identification de la chaîne de compilation fournit une aide puisque la recherche dans des programmes générés par la même chaîne de compilation est plus facile [38,136]. Par exemple, l'identification de la chaîne de compilation avec Vestige [79] améliore la précision de la recherche d'une fonction binaire avec Gemini de 21 points sur des programmes venant des paquets Coreutils et Binutils.

Investigation numérique. La chaîne de compilation permet de nourrir le champ des investigations numériques. Par exemple, la chaîne de compilation d'un programme malveillant donne des indices sur les auteurs. Or, la découverte de l'identité des auteurs peut être un enjeu géopolitique, l'attribution du programme malveillant Babar en atteste. Plusieurs indices suggérant une origine française du logiciel ont été mis en lumière par le service de renseignement canadien. Parmi ceux-ci, le choix de l'unité du kilooctet et non du kilobyte, la présence de `fr_FR`



(a) Code source. (b) GCC 4.8.5 -O0. (c) CFG. (d) GCC 9.3.0 -O0. (e) CFG.

FIGURE 2.1 – Deux chaînes de compilations différentes compilant un code source.

dans le code, et l'usage du français ainsi qu'un anglais imparfait dans les chaînes de caractères. Mais l'élément le plus suggestif reste la découverte du nom de code interne au logiciel : Babar , tout comme la trace d'un développeur prénommé titi. Ces indices ont conduit à soupçonner que Babar avait été créé par la DGSE, le service de renseignement extérieur français, une hypothèse confirmée plus tard par Bernard Barbier, ancien directeur technique à la DGSE. Cette attribution a des implications considérables sur la scène internationale, elle met en lumière les attaques de la France contre des alliés proches. En effet, bien que Babar ait fortement ciblé l'Iran, il a également ciblé plusieurs pays européens et d'anciennes colonies françaises, ainsi que des organisations francophones, y compris des médias. Dans un tout autre registre, les paquets libres peuvent être vulnérables à une exploitation commerciale non autorisée. Par exemple, l'association 'GPL Violations project' a réussi à prouver l'utilisation inappropriée d'une portion du noyau Linux par l'entreprise allemande D-Link en 2006. En identifiant la chaîne de compilation utilisée pour un produit suspecté de plagiat, on peut compiler le code source protégé avec celle-ci. S'il y a violation des droits d'auteur, le produit contrefait devrait être identique au programme obtenu.

Exemple La Figure 2.1 présente les différences induites par la chaîne de compilation, plus précisément la version du compilateur. Le code source comporte une boucle contenant une condition. La première chaîne de compilation est la version 4.8.5 de la famille de compilateurs GCC avec le niveau d'optimisation minimal -O0. La seconde chaîne de compilation est une version plus récente du compilateur, la version 9.3.0. L'architecture matérielle des deux chaînes est Intel x86-64. Les programmes produits sont en partie différents puisque les graphes de flot de contrôle sont différents. En observant attentivement le code assembleur, nous nous rendons compte que les différences concernent les trois premières instructions de la section L2 qui ont été transférées au début de la section L4.

2.1.1 Problématique

La plupart des approches existantes [32, 105, 121, 126, 143] s'intéressent à retrouver la chaîne de compilation de chaque fonction binaire séparément. En effet, il est fréquent qu'un programme soit l'union de différentes parties ayant différentes chaînes de compilations. Cependant, les fonctions binaires sont de très petits objets. À ce niveau de détail, l'influence d'une optimisation n'est pas détectable dans toutes les fonctions. À l'opposé, le flot global du programme est très influencé par les options de compilations. Par exemple, l'extension inline d'une fonction bouleverse l'exécution [76]. On peut également citer des opérations de refactorisation, telles que l'élimination de code inutile et la propagation de constante comme exemple d'optimisations au niveau du programme. Pour les nombreux programmes compilés avec une unique chaîne de compilation, une alternative globale qui détermine une unique chaîne de compilation est nécessaire.

De plus, les capacités d'IDA [5] à prédire la famille du compilateur ayant généré des programmes sont limitées. Certes, sur les 15 programmes comportant des symboles, IDA distingue correctement les programmes venant de la famille de compilateurs MinGW [6] des programmes venant de la famille de compilateurs Microsoft Visual Studio. Cependant, ce n'est pas le cas pour les programmes dépouillés de symboles qui sont tous classifiés comme venant de la famille Visual Studio.

Objectif *Notre objectif est de mettre au point une méthode automatique de prédiction d'une unique chaîne de compilation sur des programmes dépouillés de symboles.*

Défis Une telle identification de la chaîne de compilation présente plusieurs défis. Premièrement, la méthode d'identification doit pouvoir relever de subtils motifs de chaînes de compilation puisque les symboles ont été retirés. Deuxièmement, il existe une multitude de chaînes de compilation puisque la chaîne de compilation est la combinaison de la famille du compilateur, de sa version et des options de compilations utilisées. La méthode doit pouvoir s'adapter à des centaines de combinaisons. Enfin, les programmes sont des objets de tailles importants qui sont de plus très variables, la méthode doit pouvoir supporter à la fois la variabilité et la masse des données à analyser. En résumé, il est difficile d'obtenir et de pouvoir exploiter un jeu de données suffisamment diversifié qui couvre beaucoup de chaînes de compilation et types de programmes pour pouvoir identifier la chaîne de compilation d'un programme dépouillé de symboles.

2.2 État de l'art

Rosenblum et al. dans deux articles pionniers [124, 126] ont été les premiers à tenter de récupérer le compilateur et les options du compilateur. Ils ont inventé un réseau SVM où les entrées sont composées d'expressions régulières (*idioms*) sur le programme d'assemblage ainsi que de petits graphes de trois sommets.

5. Du moins, son édition gratuite.
6. MinGW est l'adaptation du compilateur Linux GCC à la plateforme Windows.

Bien qu'ils obtiennent une précision et une exactitude excellentes, ils considèrent un ensemble de données plutôt restreint en matière de diversité de codes sources, de familles de compilateurs, de versions et d'options. Il y a 175 codes sources différents. Seules 9 versions de compilateurs sont prises en compte et le niveau d'optimisation est considéré soit comme faible (p. ex. O0), soit comme élevé (p. ex. O3).

Rahimian et al. [121] ont conçu BinComp, avec un modèle complexe basé sur trois couches, la dernière utilisant un CFG annoté. Ils adoptent une approche stratifiée : d'abord, la famille du compilateur est devinée, puis, étant donné celle-ci, le niveau d'optimisation est déterminé. Avec une combinaison de caractéristiques provenant du graphe de flot de contrôle et des séquences d'instructions, la chaîne de compilation est attribuée par des noyaux de graphes utilisant du hachage ainsi que des empreintes digitales. Ce modèle nécessite la présence de symboles.

Plus récemment, quatre articles ont été publiés sur la prédiction de la chaîne de compilation. Yang et al. [159] extraient 1024 bits et les traitent avec un CNN unidimensionnel. Otsubo et al. [116] extraient une séquence de 16 instructions et agrémentent un CNN unidimensionnel d'une couche d'attention. Chen et al. [32] ont conçu Himalia, qui est un classificateur à deux couches de neurones. Les caractéristiques sont extraites de fonctions binaires et consistent en une séquence de types d'instructions de taille fixe. Ainsi, Himalia se concentre sur le prologue et l'épilogue des fonctions, et par conséquent, peut expliquer les décisions. Cela dit, les auteurs ont fait l'hypothèse forte de pouvoir déterminer le prologue et l'épilogue des fonctions. Tial et al. [143] ont appliqué une vectorisation sur les instructions binaires normalisées. La normalisation a pour but de retirer des détails afin d'augmenter les capacités de généralisation des classifieurs. Par exemple, l'instruction `mov eax, [ 0×ff5c2a8 ]` est transformée en `mov eax, MEM`. Nous nous sommes inspirés de la normalisation des instructions dans la fabrication de notre CFG réduit. La suite d'instructions est traitée par un CNN unidimensionnel agrémenté d'une couche basée sur l'attention.

La conception de méthodes de vectorisation d'instructions est un sujet en soi. Différentes méthodes ont été adaptées depuis le cadre du traitement automatique des langues, comme Asm2Vec [145], ou une vectorisation unique pour des instructions de différentes architectures [122] ou I2V [105].

Enfin, deux travaux sont proches de notre travail. Le premier travail est celui de Massarelli et al. [105] en 2019 que nous évaluons dans la Section 2.7.4. Là où nous ciblons les programmes entiers, ce travail se focalise sur la provenance des fonctions binaires, et ils extraient les CFG des fonctions. Sur ceux-ci, ils utilisent une vectorisation des graphes. La phase d'apprentissage est composée de deux étapes. La première étape transforme les séquences d'instructions en blocs de base en utilisant une méthode de vectorisation des instructions appelée I2V et un réseau de neurones récurrent. Ensuite, des convolutions sur le graphe fournissent un vecteur. Le second est Vestige [79] de Ji et al. sorti en juin 2021 soit quelques mois après notre travail. Il s'intéresse également à la chaîne de compilation d'un programme entier et non d'une fonction. La principale différence se trouve dans l'objet utilisé pour classifier les programmes. Nous utilisons un graphe de flot de contrôle du programme entier réduit tandis que Vestige utilise le graphe d'appels de

fonctions, en intégrant des attributs venant des CFG des fonctions. Ils considèrent deux types de caractéristiques : des séquences d'instructions et des petits graphes venant des CFG des fonctions. Comme dans de nombreux travaux déjà présentés, les instructions sont normalisées. Une petite partie des caractéristiques fait office d'attribut des sommets du graphe d'appels de fonctions. En raison du grand nombre de caractéristiques possibles, les caractéristiques ont une taille maximale de trois. Ils appliquent ensuite une méthode de vectorisation sur le graphe d'appels de fonctions attribué.

2.3 Contributions

Nous affirmons que le processus de décision peut être effectué au niveau du *programme* binaire plutôt qu'au niveau de la *fonction* binaire comme dans la plupart des approches antérieures. En nous éloignant des sémantiques précises de blocs de base, nous suggérons d'utiliser un graphe de flot de contrôle (CFG) *réduit* du programme binaire entier. C'est pourquoi nous suggérons d'utiliser des *réseaux de neurones sur les graphes* (GNN) [163]. En conséquence, les relations entre les blocs de base, les nœuds du CFG, sont prises en compte dans la vectorisation du graphe et le poids d'un nœud CFG dépend transitivement de ses voisins.

En agissant ainsi, nous partons du principe que le programme n'est pas comme un texte ou une image qui peut être projetée dans un espace euclidien régulier, et par conséquent il est utile de conserver la structure du programme, du moins partiellement.

Nous apportons les contributions suivantes.

Première contribution Nous développons un modèle basé sur les GNN que nous appelons *Site Neural Network* (SNN) pour déterminer la famille du compilateur, la version du compilateur et le niveau d'optimisation qui génèrent un code binaire donné. L'architecture globale est affichée dans les Figures 2.2 et 2.3. Nous extrayons un CFG complet à partir d'un code binaire et le réduisons en conservant seulement son squelette. Ensuite, ce graphe *réduit* est *découpé* en un ensemble de sous-graphes de taille fixe (nommés *site*) dont la taille est un paramètre noté α dans la suite du texte. Cette étape de prétraitement est entièrement automatique et non supervisée. Ensuite, le réseau de neurones prend le graphe de tous les sites en entrée afin de classer le code binaire. L'architecture globale du réseau adapte le réseau à résidus (ResNet) aux sites [162], tout en affinant ce modèle avec des couches adaptatives de regroupement par les maximums (AMP) [135]. Nous proposons également de former des hiérarchies de plusieurs classifieurs locaux prenant des décisions binaires, par exemple décider si un programme est compilé avec «Clang 8.0» ou «Clang 8.5».

Notre approche présente au moins trois avantages.

1. Comparé aux travaux antérieurs [32, 105, 121, 126, 159], notre modèle est assez simple, car il se contente d'analyser des sites. Il n'y a pas de caractéristiques d'instruction ni de focalisation sur les prologues ou les épilogues de fonctions. Nous nous attendons donc à ce que SNN soit plus robuste et générique ;

2. D'un point de vue méthodologique, notre cadre de travail fournit une solution de bout en bout basée sur des classificateurs de graphes. Par conséquent, les décisions et les classifications devraient être plus facilement basées sur les sémantiques du code binaire ;

3. Notre cadre de travail peut être facilement adapté à différents contextes : le découpage est paramétré par la taille du sous-graphe α, tandis que notre hiérarchie permet d'ajouter de nouveaux classifieurs locaux de manière modulaire – par exemple pour considérer une nouvelle version de compilateur ou une nouvelle option.

Seconde contribution La plupart des travaux précédents sur la provenance des outils basés sur l'apprentissage automatique utilisent comme jeu de données un ensemble de *fonctions* binaires générées par différents compilateurs et niveaux d'optimisation. De notre point de vue, cette approche crée un biais, qui peut être acceptable selon le contexte, ou pas. En effet, il peut être difficile d'identifier correctement le début et la fin des fonctions lorsque nous traitons des binaires offusqués ou dépouillés et le temps de traiter toutes les fonctions, par exemple dans les COTS, peut être important. De plus, la plupart du temps, les bibliothèques, comme les DLL, sont liées dynamiquement et il n'est donc pas nécessaire de déterminer la chaîne d'outils de compilateur pour chaque fonction individuellement. *C'est pourquoi nous nous concentrons sur les binaires.* Par conséquent, notre ensemble de données consiste en des binaires complets dépouillés sans aucune connaissance des fonctions et de leur localisation.

Troisième contribution Certaines études souffrent d'une variété limitée dans leurs jeux de données. Par exemple, seulement 18 configurations de compilateur sont étudiées par Rosenblum et al. [124]. Notre étude couvre un grand nombre de chaînes de compilation possibles sur Linux et sur Windows pour un système 64 bits. En effet, l'identification couvre 23 versions différentes de quatre compilateurs majeurs : Clang, GCC, MinGW et Visual Studio. Nous prenons en considération quatre optimisations (-O0, -O1, -O2/-O3 et -Os) pour un total de 92 configurations de compilateurs. Nous avons évalué notre système en termes de précision de détection sur un vaste jeu de données composé d'environ $36,272$ binaires dépouillés compilés à partir de $36,272$ codes sources différents avec quatre familles de compilateurs, 23 versions différentes de compilateurs et cinq niveaux d'optimisation. Ainsi, nous pouvons entraîner et tester notre approche avec 92 ensembles distincts et équilibrés de codes binaires, ayant tous des codes sources différents. Nous démontrons que l'identification de la provenance des outils *au niveau du programme* est faisable avec une bonne précision tout en conservant une phase d'apprentissage efficace. Nous obtenons de meilleurs résultats qu'une méthode récente classifiant les fonctions [105] malgré un temps d'apprentissage beaucoup plus court (voir la Section 2.7.4). Dans l'ensemble, nous croyons que ces résultats sont prometteurs et peuvent offrir de nouvelles pistes pour l'identification de la provenance des outils.

2.4 Méthodologie

Puisque nous avons accès à énormément de données, notre méthode de prédiction utilise un apprentissage supervisé. Ce paradigme nécessite de réserver une large partie des données à une phase d'entraînement. Durant l'entraînement, les chaînes de compilation des programmes sont connues et notre méthode apprend à prédire celles-ci. Parmi les époques de l'entraînement, nous sélectionnons le modèle statistique avec la meilleure prédiction sur un sous-ensemble du jeu d'entraînement réservé à la sélection. Nous utilisons le critère de la proportion de prédictions correctes dans cet ensemble.

Les métaparamètres, qui guident l'ensemble du comportement d'un algorithme d'apprentissage comme le taux d'apprentissage, ne sont pas appris automatiquement, mais doivent être réglés préalablement. Leur ajustement est essentiel pour produire un bon modèle.

Notre méthode de prédiction d'une chaîne de compilation doit être évaluée sur un ensemble de programmes. Cet ensemble doit contenir différentes chaînes. Une grande variété de programmes doit pouvoir être traitée par la méthode.

Le reste des données est utilisé après l'entraînement et la sélection pour évaluer la méthode. Les critères d'évaluation sont la précision et le rappel (voir la Section 1.3.1) pour des chaînes de compilation ou des groupes de chaînes de compilation.

Par exemple, on peut regrouper les programmes ayant un niveau d'optimisation -O0 dans une classe notée -O0. La précision P est la proportion de programmes dans la classe -O0 parmi les programmes prédits par la méthode comme faisant partie de la classe -O0. Le rappel R est la proportion de programmes ayant été prédits comme faisant partie de la classe -O0 parmi les programmes dans la classe -O0. Pour rendre compte des performances en une métrique, nous mesurons le F1-Score qui est égal à $2 \times \frac{P \times R}{P+R}$. Nous mesurons la précision, le rappel et le F1-Score de chaque famille de compilateurs, chaque niveau d'optimisation, et chaque version de compilateur. Notre attention se porte non seulement sur la qualité des prédictions, mais également sur le temps d'apprentissage requis par nos modèles.

Avant de décrire les résultats de ces expériences, nous allons commencer par expliquer notre méthode et la constitution de notre jeu de données.

2.5 SNN, un nouveau classifieur de graphe

2.5.1 Les réseaux de neurones sur les graphes

Pour prédire les chaînes de compilation, nous avons choisi de disséquer les graphes de flot de contrôle. Nous souhaitons obtenir une représentation des CFG qui met en relief la chaîne de compilation. Pour cela, nous construisons une empreinte du graphe qui contient une trace de la chaîne de compilation.

L'étude de Hamilton et al. [60] discute des différentes représentations possibles. Contrairement aux données non structurées comme les textes ou les images, les opérations sur les graphes doivent être invariantes aux isomorphismes. En s'inspirant du test de Weisfeiler-Lehman (voir la

Section 1.4), on peut construire un réseau de neurones sur les graphes (GNN) [107, 132, 163]. En effet, l'agrégation des couleurs correspond à une étape de convolution. Les réseaux de neurones sur les graphes par convolution sont nommés GCN. À chaque étape, le GCN agrège simplement la valeur de chaque sommet avec les voisins. L'agrégation est le plus souvent une moyenne, sous la forme : $D^{-1} \times A \times Z_k \times W_k$ où V_k est la valeur des sommets à l'étape k, A est la matrice d'adjacence, D est la matrice des degrés et W_k est une matrice à apprendre par descente de gradient.

Utilisation de résidus On peut également incorporer les idées du réseau à résidu ResNet [63] ou DenseNet [71] voir les travaux de Zhao et al. [162]. Ces réseaux font passer le résultat d'une convolution i à la convolution $i + 2$. Cette idée simple suffit à améliorer les capacités du réseau sans augmenter le nombre de convolutions.

Empreinte Tout comme durant le test de Weisfeiler-Lehman, on peut collecter les valeurs de chaque sommet à chaque étape et l'utiliser comme empreinte. Cette matrice Z est de taille $l \times n$ où n est le nombre de sommets et l est le nombre de convolutions. Cependant, cette matrice n'a pas une dimension fixe, puisque les graphes n'ont pas tous la même taille. On doit alors introduire une couche de regroupement, où on réduit Z à une taille fixe.

Regroupements Le regroupement le plus simple est la somme. En faisant la somme des valeurs des sommets, on obtient un vecteur de taille l. On peut également utiliser une moyenne. Un article de Zhang et al. [161] propose une méthode de regroupement par le tri. Cela consiste à sélectionner un nombre fixe k de sommets aux valeurs les plus grandes. Ainsi, on obtient un vecteur de taille $k \times l$. Enfin, un article de Junhyun et al. [96] propose de conserver un nombre prédéfini k_i de sommets à chaque étape de convolution i. On obtient alors une matrice de taille $\sum k_i \times n$. L'élimination progressive modifie le comportement de la phase d'apprentissage par rapport à une sélection a posteriori.

2.5.2 Le découpage en sites

Les graphes de flot de contrôle contiennent des milliers de sommets. Ce graphe a un grand diamètre, ce qui implique qu'il faut un grand nombre de convolutions pour que l'information se propage à travers l'intégralité du graphe. Pour pallier ce problème, nous allons découper le graphe de flot de contrôle d'un programme en un ensemble de petits graphes de tailles fixes. Ces différents *sites* seront réunis par la phase de regroupement.

Les entrées sont des programmes encodés en binaires. Le graphe de flot de contrôle (CFG) est extrait avec le désassembleur Gorille [7] utilisant une symbolique concrète pour parcourir tous les chemins d'exécution possibles du programme [22]. Il y a ensuite deux étapes, la phase de

7. `https://cyber-detect.com/gorille/`

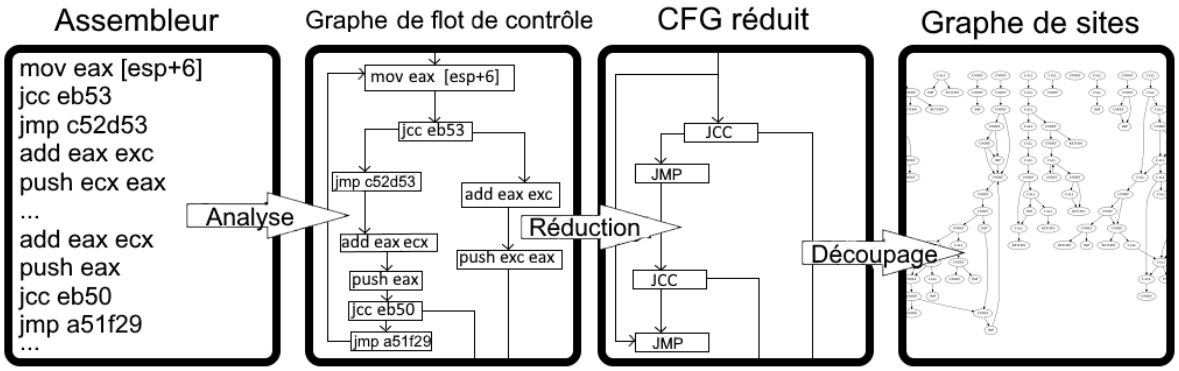


FIGURE 2.2 – Déroulement du prétraitement.

simplification et la phase de découpage. Ces phases réduisent drastiquement le graphe et facilitent la transmission de l'information durant la phase de convolution.

La phase de simplification retire les instructions séquentielles du CFG en gardant les instructions concernant le flot de contrôle. Cette phase a deux étapes :

1. Tous les sommets consécutifs qui sont des instructions séquentielles (telle que `mov` ou `add`) sont retirés ;
2. Tous les sommets restants sont attribués selon leur type d'instruction d'après la Table 2.1.

TABLE 2.1 – Attribution automatique.

Symbole	Signification
RET	retour de fonction
CALL	appel de fonction
JMP	saut inconditionnel
HLT	interruption
INVALID	échec de désassemblage
UNDEF	adresse inconnu
JCC	saut conditionnel
SWITCH	saut à plusieurs destinations

La phase de réduction respecte la structure sous-jacente du CFG et produit le *CFG réduit*.

La phase de découpage coupe le CFG réduit en un nombre fixe de petits graphes. Ces petits graphes sont appelés *sites* et ont une taille de maximum α sommets. Les sites sont obtenus par une exploration en largeur depuis le CFG réduit. Cette exploration part d'une racine du CFG réduit, traditionnellement le point de départ du programme, et s'arrête après avoir collecté un certain nombre de sites. L'algorithme est présenté dans l'algorithme 2.

Algorithm 2 Algorithme de découpage de graphe.

Entrées : $G = (V, E)$, un CFG réduit et r, un sommet racine de G

Paramètres : n, le nombre de sites à extraire et α, la taille maximum d'un site

Sortie : G_o, un graphe contenant un maximum de n sites

1: Soit $G_o = (V_o, E_o)$ un graphe
2: **Tant que** $|V| > 0$ **et** $n > 0$ **Faire**
3: Soit $q1$ une pile
4: Soit $q2$ une pile
5: Soit $g = (N, A)$ un graphe
6: **Empiler** r **sur** $q1$
7: **Tant que** $|N| < \alpha$ **et** $|q1| > 0$ **Faire**
8: **Vider** $q2$
9: **Pour tout** $x \in q1$ **Faire**
10: **Si** $|N| \geq \alpha$ **Alors**
11: **Sortir**
12: **Fin Si**
13: **Pour tout** y **tel que** $(x, y) \in E$ **Faire**
14: **Si** $|N| \geq \alpha$ **Alors**
15: **Sortir**
16: **Fin Si**
17: **Si** $y \in$ N **Alors**
18: $A \leftarrow A \cup \{(x, y)\}$
19: **Sortir**
20: **Fin Si**
21: $N \leftarrow N \cup \{y\}$
22: $A \leftarrow A \cup \{(x, y)\}$
23: **Empiler** y **sur** $q2$
24: **Fin Pour**
25: **Fin Pour**
26: $q1 \leftarrow q2$
27: **Fin Tant que**
28: $V \leftarrow V \setminus \{N\}$
29: $E \leftarrow E \setminus \{(u, v) | u \in N \text{ ou } v \in N\}$
30: $r \leftarrow$ **racine**(G)
31: $V_o \leftarrow V_o \cup N$
32: $E_o \leftarrow E_o \cup A$
33: $n \leftarrow n - 1$
34: **Fin Tant que**
35: **Renvoyer** G_o

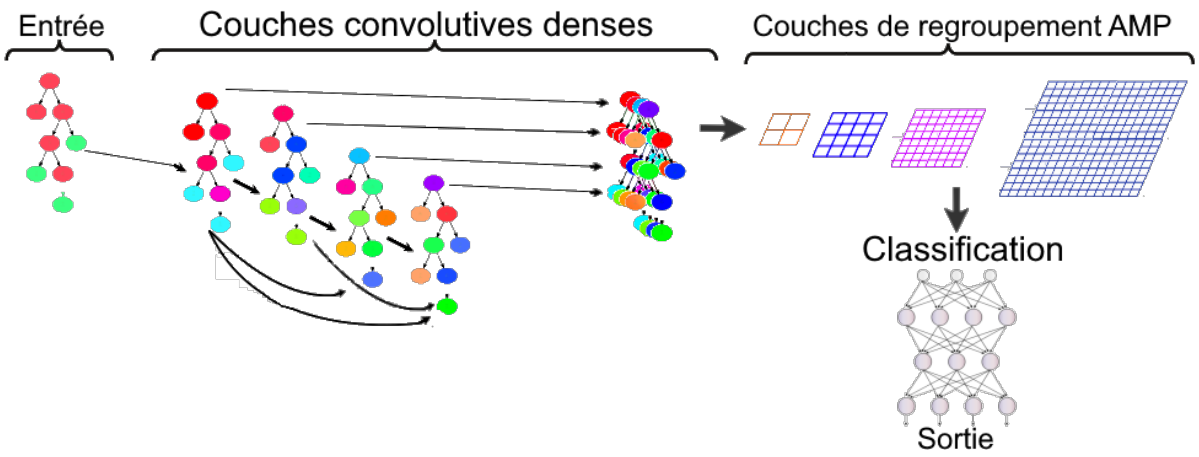


FIGURE 2.3 – Architecture d'un réseau de neurones sur les sites.

2.5.3 Les Site Neural Network (SNN)

Chaque sommet d'un site a deux caractéristiques. Premièrement, le type d'instruction (voir la Table 2.1) et deuxièmement un identifiant unique. Les sites ont de petits diamètres d'au plus α, un petit nombre de convolutions permet alors de faire passer l'information à travers tous les sommets. Les sites sont des graphes dirigés, cependant, ils sont traités comme non dirigés par l'opération de convolution. Une entrée du réseau de neurones est un ensemble de sites. Ces sites sont rassemblés en un graphe dont les sites sont les composantes connexes. Soit V_r, l'ensemble des sommets de ce graphe et A sa matrice d'adjacence. Nous définissons X_0 la matrice contenant les caractéristiques des sommets, elle a pour dimension $|V_r| \times 2$.

Mini-batchs Durant la phase d'apprentissage, plusieurs graphes sont réunis en un seul mini-batch. Cela permet de normaliser les entrées [75]. Un mini-batch B est normalisé par la fonction $\text{batchNorm}_B(x) = \frac{x-\mu_B}{\sigma_B}$, où μ_B est la moyenne observée et σ_B est la variance observée. Ces observations sont mémorisées et réutilisées durant la phase de test. Ce processus est concluant, mais n'a pas encore été compris théoriquement.

Convolution dense La fonction d'activation suivant une convolution est $\text{relu}(x) = \max(0, x)$.

Le vecteur des caractéristiques $(Y_{k+1})_{k\geq 0}$ obtenu après $k+1$ convolution(s) est défini par :

$$Y_1 = \text{relu}(\text{batchNorm}_B((A+I)X_0W_0 + b_0))$$
$$Y_{k+1} = (\text{relu}(\text{batchNorm}_B((A+I)Y_kW_k + b_k))|Y_k)$$

La notation | est la fusion de matrice. En utilisant le concept de convolution dense introduite par Huang et al. [71], la sortie d'une étape est fournie à toutes les étapes suivantes.

Dimensions Soit d_t le paramètre de la seconde dimension de la matrice W_t à la t ème convolution. La première dimension de la matrice W_k, pour $k > 0$, est $\sum_{t=0}^{k-1} d_t$.

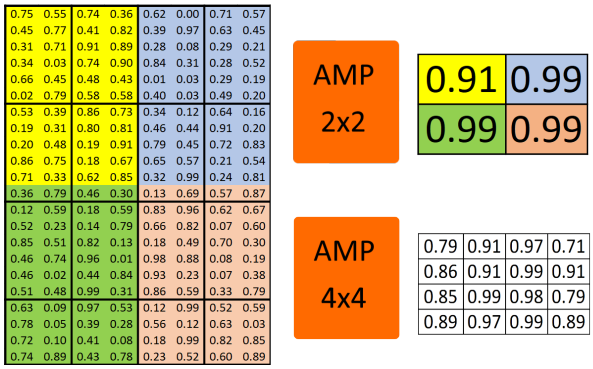


FIGURE 2.4 – Deux regroupements par couche AMP sur une matrice de dimension 22×8. La première couche AMP produit une matrice de taile 2×2. La seconde produit une matrice de taille 4×4.

Sortie La matrice Y_k, pour $k > 0$, a comme dimension $n \times \sum_{t=0}^{k} d_t$ où n est le nombre de sommets. Nous devons maintenant effectuer le regroupement pour réduire cette matrice à une plus petite dimension de taille fixe. Mais tout d'abord, il nous faut séparer le mini-batch par rapport aux entrées d'origines. En effet, les opérations suivantes ne peuvent être appliquées sur plusieurs graphes en même temps.

Regroupement Nous appliquons une couche de regroupement AMP. C'est le regroupement inventé par Jiaqi et al. [157]. Cela consiste à regrouper par des maximums Y_k en une matrice de taille $a \times b$. L'opération $\text{amp}_{a,b}$ procède de la façon suivante : Y_k est découpé en $a \times b$ matrices de tailles $\lceil \frac{u}{x} \rceil \times \lceil \frac{v}{y} \rceil$. Puis, le maximum est sélectionné dans chaque bloc. La Figure 2.4 décrit une couche AMP.

Classification Il nous faut faire passer l'empreinte dans un réseau de neurones à plusieurs couches pour prédire la classe de l'entrée. La dernière couche calcule la distribution de probabilité de la classe de l'entrée.

Paramètres Nous réunissons 20 entrées en un mini-batch. Nous effectuons quatre convolutions, la matrice W_t a une dimension d_t de 8 pour chaque convolution. De même, nous utilisons quatre couches d'AMP ($\text{amp}_{2,2}$, $\text{amp}_{4,4}$, $\text{amp}_{8,8}$, et $\text{amp}_{16,16}$). Enfin, le réseau de neurones final comporte également quatre couches. Leurs nombres de neurones respectifs sont 384, 256, 128, et 2. Chaque SNN a 286 962 paramètres à apprendre. Le taux d'apprentissage est de $0,005$ et la fonction objectif est l'entropie croisée.

Nous implémentons notre réseau de neurones en Python avec la bibliothèque PyTorch. Le prétraitement est effectué en C++.

Les données et outils sont disponibles à l'adresse suivante :

`https://gitlab.inria.fr/tbenoit/saner-2021-binary-level-toolchain-provenance-identification.`

2.5.4 Les hiérarchies

Les travaux précédents [32, 105, 124, 126, 159] considèrent chaque tâche, telle que déterminer le niveau d'optimisation, séparément, et utilisent un seul réseau de neurones pour classifier au niveau des fonctions binaires.

Au contraire, la dernière couche d'un SNN a une sortie de dimension 2. Nos SNN font des choix binaires entre deux ensembles. Pour prédire la chaîne de compilation, nous utilisons plusieurs SNN. Par exemple, le premier SNN décide si le programme fonctionne pour la plateforme Linux ou Windows. Nous implémentons donc des hiérarchies de classifieurs binaires avec un classifieur local par nœud parent [137]. Un classifieur local est spécialisé pour séparer deux ensembles de classes (p. ex. entre MinGW -O0/-O1 et MinGW -O2/-O3). De plus, nous prenons l'avantage de la connaissance de notre problème, par exemple, nous savons que les instances compilées avec -O0 et -O1 sont plus proches que les instances compilées avec -O0 et -O2/-O3. De même, la prédiction de la famille du compilateur est utile à d'autres prédictions.

2.6 Jeu de données CodeForces

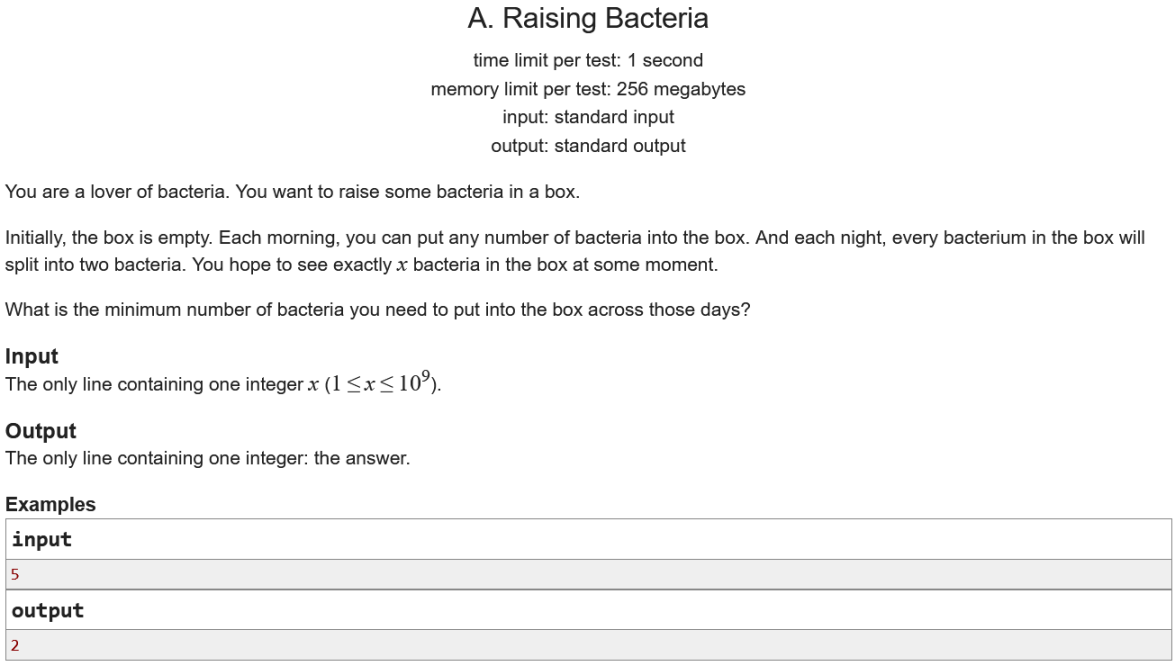
Notre jeu de données provient du site internet CodeForces. Ce site propose de résoudre des problèmes donnés en écrivant des programmes. Un jeu de données provenant du domaine de la résolution de problème a déjà été utilisé au préalable par Phan et al. [145] pour la détection de défaut par le graphe de flot de contrôle. L'origine du jeu de données provient de l'extraction par Alexey Grigorev de 273 705 codes sources écrit sur le site [8].

Les données contiennent pour chaque code source le résultat de l'application sur le problème demandé. Un exemple de problème [9] est donné en Figure 2.5. Une solution valide de ce problème est disponible en Figure 2.1. Un code source peut être accepté, ou refusé à la suite d'un échec sur une entrée préétablie. Certains codes sources plantent à l'exécution tandis que d'autres ne peuvent pas être compilés. Afin de couvrir le plus de codes sources, nous ne souhaitons pas nous restreindre à ceux qui résolvent le problème. Cependant, les 9704 codes sources que CodeForces n'a pas réussi à compiler ne nous intéressent pas.

Les données contiennent également le langage de programmation ainsi que la plateforme. Nous découvrons que 245 248 codes sources sont écrits dans le langage C ou dans son évolution le C++. Nous nous concentrons sur des programmes compilés depuis ce langage. Nous remarquons que plus de 220 000 sont écrits en C++ pour la plateforme Linux tandis que moins de 17 000 sont écrits pour la plateforme Windows. De même, moins de 9000 sont écrits en C, le reste étant écrit en C++. Ce déséquilibre doit être corrigé afin de couvrir au maximum les différentes chaînes de compilations.

Ces codes sources couvrent 91 problèmes de CodeForces différents. Étant donné le nombre très important de codes sources et le faible nombre de problèmes, nous souhaitons un nombre

8. https ://www.itshared.org/2015/12/codeforces-submissions-dataset-for.html
9. https ://codeforces.com/problemset/problem/579/A



A. Raising Bacteria

time limit per test: 1 second
memory limit per test: 256 megabytes
input: standard input
output: standard output

You are a lover of bacteria. You want to raise some bacteria in a box.

Initially, the box is empty. Each morning, you can put any number of bacteria into the box. And each night, every bacterium in the box will split into two bacteria. You hope to see exactly x bacteria in the box at some moment.

What is the minimum number of bacteria you need to put into the box across those days?

Input
The only line containing one integer x ($1 \le x \le 10^9$).

Output
The only line containing one integer: the answer.

Examples

input
5

output
2

FIGURE 2.5 – Le problème 579A de CodeForces.

équitable de codes sources résolvant chaque problème. Nous appliquons une sélection parmi les codes sources en essayant de limiter le nombre de codes sources portant sur le même problème à 700 et le nombre de codes sources écrit dans le même langage à 13 000. Finalement, nous obtenons 42 827 codes sources où 13 022 sont écrits pour la plateforme Windows et 29 805 pour Linux. Bien qu'une majeure partie soit écrite en C++, 5242 sont écrits en C.

2.6.1 Chaîne de compilation

Nous considérons quatre familles de compilateurs : Clang, GCC, MinGW et Visual Studio. GCC est le standard de la compilation pour la plateforme Linux tandis que Visual Studio est le standard de la compilation sur la plateforme Windows. Clang est une alternative pour Linux qui a l'avantage de proposer des avancées dans le domaine de l'optimisation des programmes par l'infrastructure LLVM. De son côté, MinGW est un portage de GCC sur la plateforme Windows. La plateforme d'exécution de GCC et Clang est Ubuntu 18.04.4 x64. Celle de Visual Studio et MinGw est Windows 10 Enterprise x64.

Nous incluons pour chaque famille de compilateurs jusqu'à six versions différentes. Au total, nous intégrons *23 compilateurs différents* : Clang 3.9.1, Clang 4.0.1, Clang 5.0.1, Clang 6.0.0, Clang 7.0.0, Clang 8.0, GCC 4.8.5, GCC 5.5.0, GCC 6.5.0, GCC 7.5.0, GCC 8.4.0, GCC 9.3.0, MinGW 3.4.5, MinGW 4.4.1, MinGW 4.7.1, MinGW 4.9.2, MinGW 5.11.0, MinGW 8.1.1, Visual Studio (VS) 10.0, VS 12.0, VS 14.0, VS 2017 et VS 2019.

Nous considérons cinq options d'optimisation : -O0, -O1, -O2, -O3, et -Os. Les quatre premières sont des niveaux optimisations de la vitesse d'exécution tandis que la dernière optimise

la taille du programme. Le compilateur Visual Studio n'implémente pas d'option -O3. Au total, il y a *120 chaînes de compilations différentes*. Nous compilons chacun de nos 42 827 codes sources avec une chaîne de compilation aléatoire. Cela garantit une bonne répartition. Une partie des codes sources ne vont pas pouvoir être compilés. Les programmes sont débarrassés de leurs symboles. Le jeu de données final contient *36 272 programmes provenant de 36 272 codes sources*.

Similarités entre -O2 et -O3 Les programmes compilés avec -O2 et -O3 sont souvent semblables. Sur un ensemble aléatoire de 2920 codes sources, 561 codes sources compilés avec MinGW et -O2 sont identiques quand compilés avec MinGW et -O3. Cela a déjà été évoqué [49] et a été un souci pour la prédiction de niveau d'optimisation de Chen et al. [32]. À cause de ces similarités entre -O2 et -O3, nous réunissons les deux options dans une classe -O2/-O3. Heureusement, dans le cas de Visual Studio il n'y a pas de niveau -O3. Cela réduit le nombre de chaînes de compilations différentes que nous devons prédire à 92.

Hiérarchies Nous utilisons deux hiérarchies différentes. La première prédit le niveau d'optimisation (Figure 2.6) et contient 15 classifieurs locaux. La seconde prédit la version du compilateur (Figure 2.7) et contient 22 classifieurs locaux. Le nombre d'étapes d'entraînement est précisé en dessous de chaque nœud sur ces Figures. Puisque trois classifieurs locaux sont en commun dans ces deux hiérarchies, nous entraînons 34 classifieurs locaux pour un total de 9 756 708 paramètres.

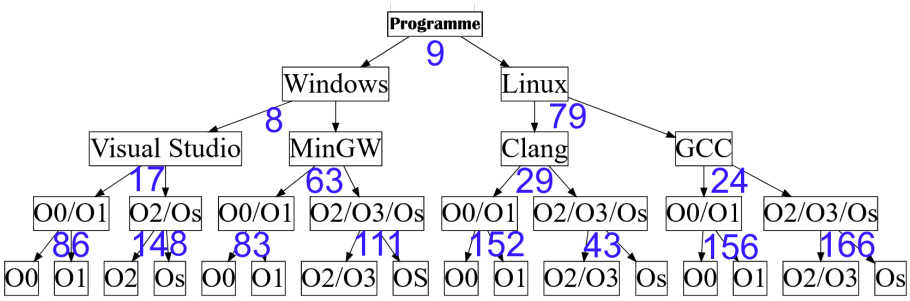


FIGURE 2.6 – La hiérarchie prédisant le niveau d'optimisation.

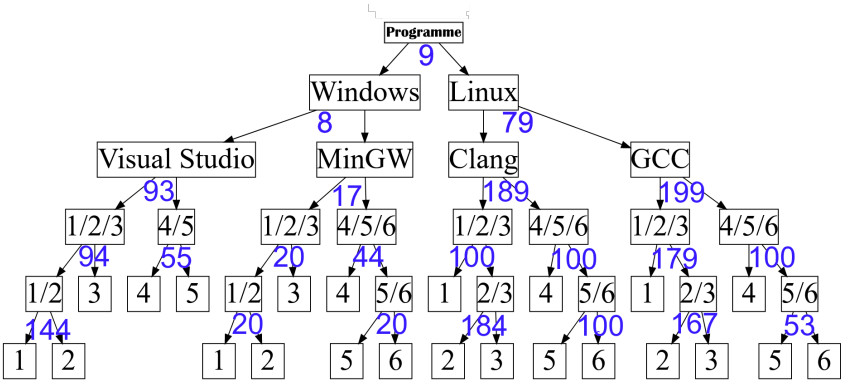


FIGURE 2.7 – La hiérarchie prédisant la version du compilateur.

2.6.2 Caractéristiques notables du jeu de données

Les 36 272 codes sources contiennent 2 766 152 lignes non vides et 51 759 411 caractères. Chaque ligne contient en moyenne 19 caractères. Un code source contient en moyenne 76 lignes et 1427 caractères. Parmi les codes sources, 35 725 sont uniques, ce qui signifie que 547 codes sources sont des doublons. Nos programmes sont donc, comme nous nous y attendions, petits, mais ils ont l'avantage d'être pour la grande majorité unique.

La Figure 2.8 illustre la distribution des langages et des plateformes des codes sources. Environ deux tiers des codes sources sont écrits pour le système d'exploitation Linux alors qu'un tiers sont écrits pour le système d'exploitation Windows.

La Figure 2.9 est un nuage de mots formé par les codes sources du jeu de données. Ce nuage de mots contient sans surprise les mots les plus usuels des langages C et C++, tels que la directive d'inclusion d'une bibliothèque `include`, le type des entiers `int`, les boucles `for`, les conditions `if`, et certaines bibliothèques (p. ex., `iostream`, `map`, et `algorithm`).

La Figure 2.10 illustre la distribution des résultats des codes sources d'après le robot de CodeForces. Seulement 11 632 codes sources sont des solutions au problème demandé. Un peu moins de 7 000 codes sources ont rencontré des problèmes divers tandis que le reste a été refusé après avoir donné une mauvaise réponse.

La Figure 2.11 donne la fréquence des problèmes résolus par les codes sources. Le problème le plus présent est celui de compter le nombre d'occurrences d'une entrée dans une table de multiplication. Un total de 1054 solutions ont été proposées. Le problème le moins présent a 39 solutions proposées et consiste à énumérer des plus larges sous séquences par programmation dynamiques.

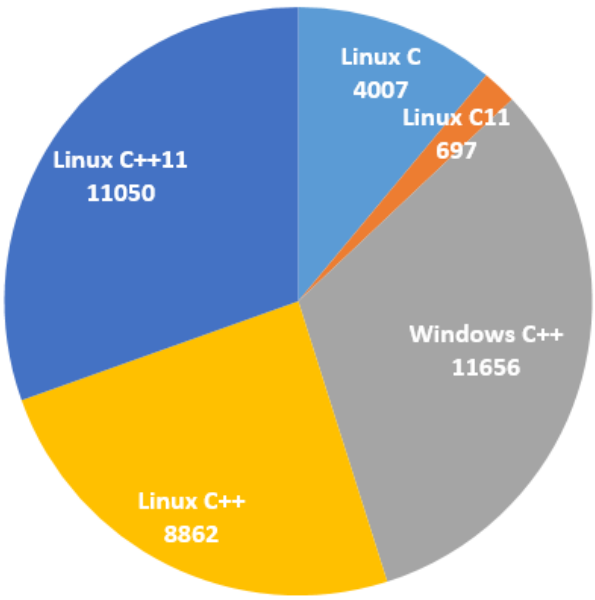


FIGURE 2.8 – Langage et plateforme des codes sources.

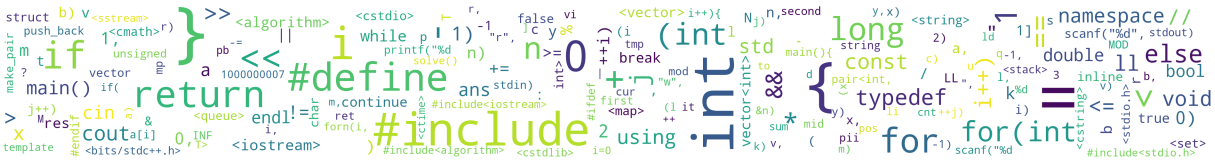


FIGURE 2.9 – Un nuage de mots d'après les codes sources du jeu de données.

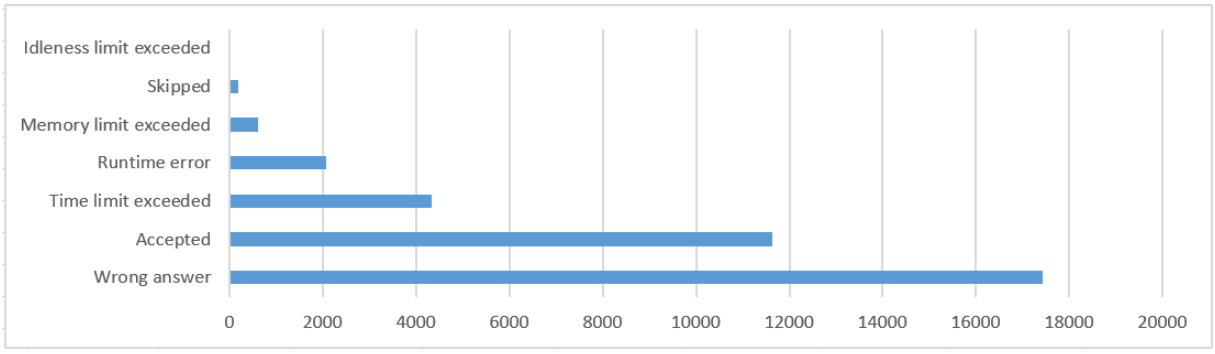


FIGURE 2.10 – Résultats des codes sources.

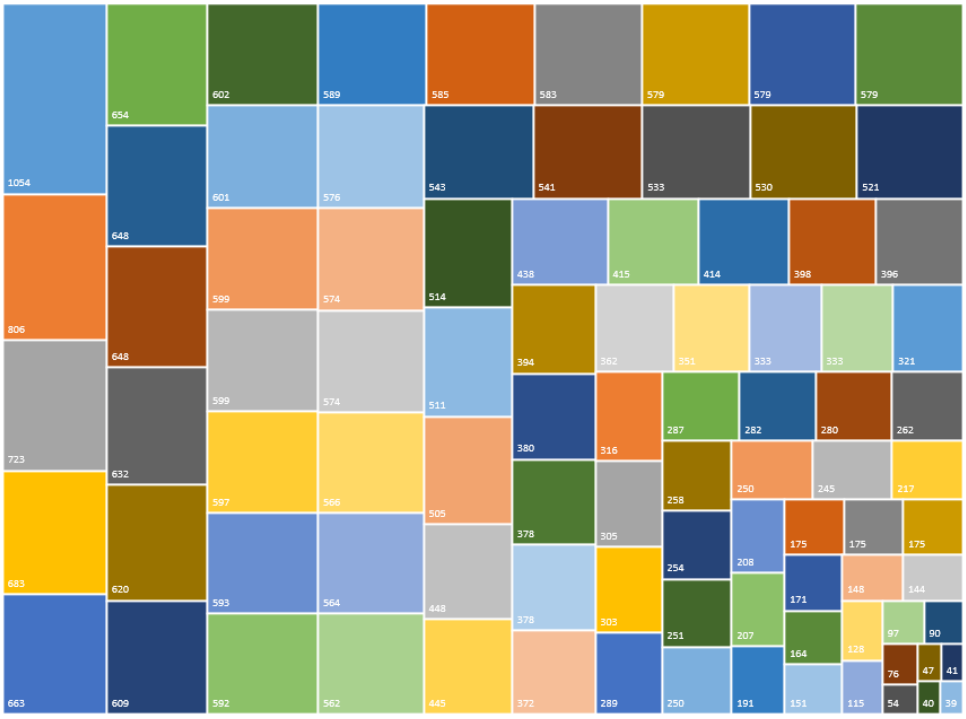


FIGURE 2.11 – Distribution des 91 problèmes.

Les caractéristiques des programmes compilés sont plus complexes à révéler. À première vue, nous savons seulement qu'un programme pèse en moyenne 135 Ko. Pour aller plus loin dans notre analyse, nous allons utiliser l'apprentissage non supervisé (voir la Section 1.3.1). Plus précisément, nous allons visualiser les données par une projection du programme dans un plan à deux dimensions. Un programme est ainsi un point sur le plan, nous pouvons donner des couleurs aux points selon les chaînes de compilations utilisées pour les produire.

Nous commençons par transformer le programme en une suite d'octets. Ensuite, nous conservons uniquement les 1000 premiers octets. Cette étape est nécessaire afin d'obtenir une dimension fixe et raisonnable. La première couche relève les 50 composantes principales [118]. Les données sont transformées de façon à conserver le maximum de variances. Ensuite, l'algorithme t-SNE [144] permet de réduire ces données à deux dimensions. Cet algorithme minimise la divergence de Kullback–Leibler [93] entre les similarités et un plan en deux dimensions.

Nous calculons des visualisations des données en colorant différemment les familles de compilateurs (Figure 2.12), les versions des compilateurs (Figures 2.13, 2.14, 2.15, et 2.16) et les niveaux d'optimisations suivant les compilateurs (Figures 2.17, 2.18, 2.19, et 2.20). Par exemple, la Figure 2.12 est une visualisation des données sur laquelle les familles de compilateurs sont visibles. Sur cette projection, les différentes familles sont bien séparées, ce qui indique que les différencier est une tâche facile. Nous remarquons tout de même de petites collusions entre les familles de compilateurs Clang et GCC. Les Figures 2.13 et respectivement 2.14 révèlent que les versions des compilateurs Visual Studio et respectivement MinGW sont relativement bien séparées.

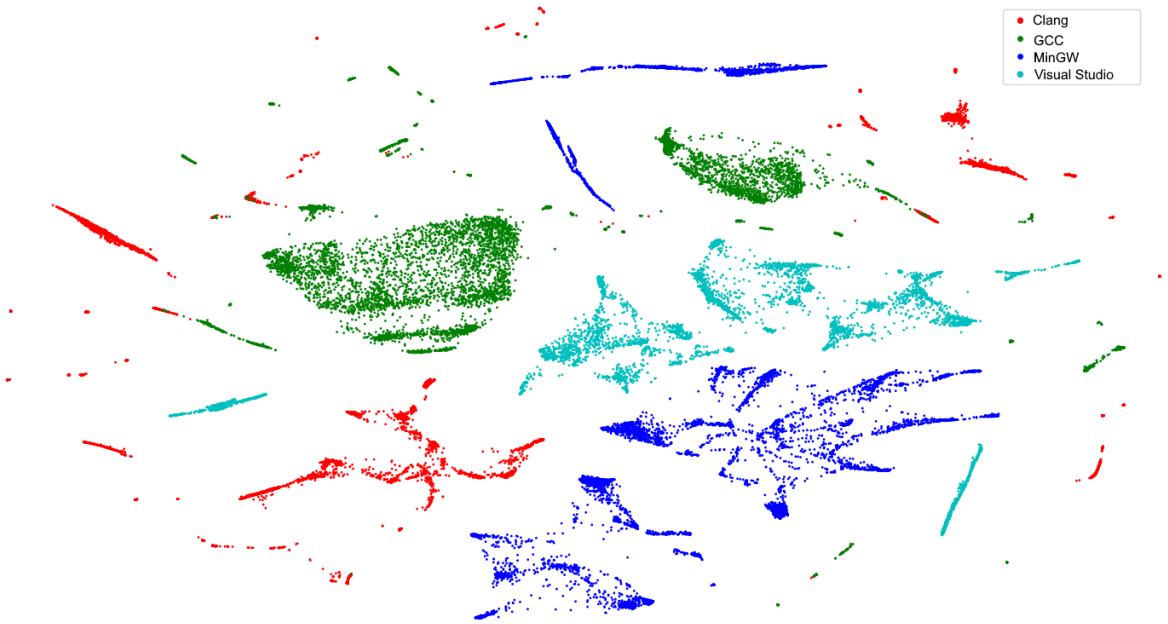


FIGURE 2.12 – Les programmes du jeu de données séparés par famille de compilateurs.

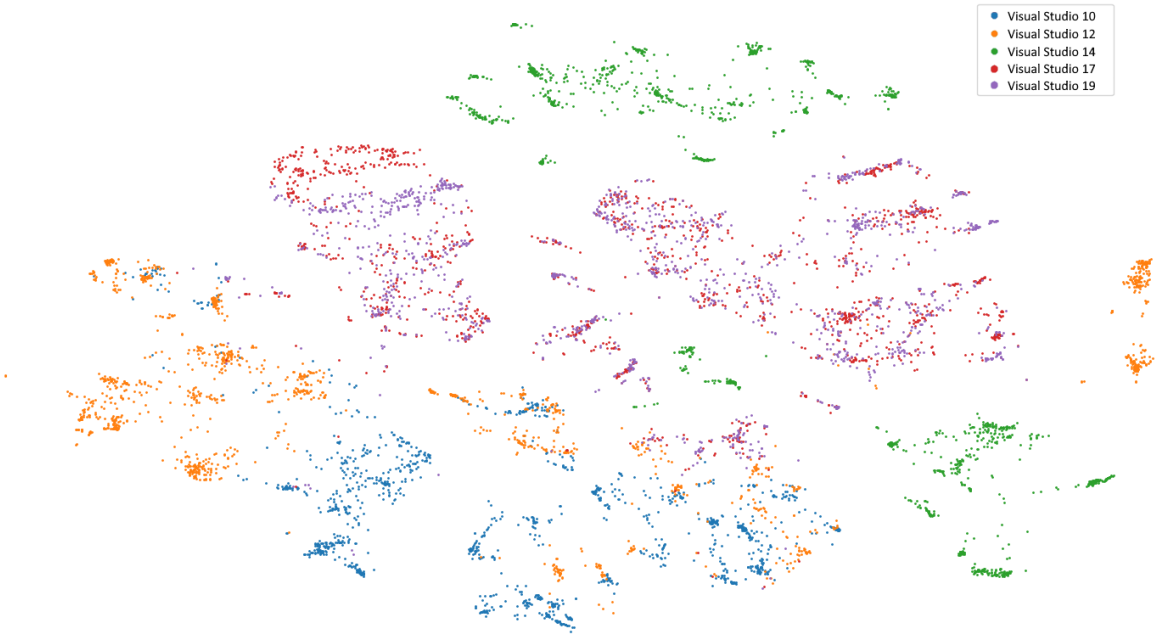


FIGURE 2.13 – Programmes compilés avec Visual Studio séparés en versions du compilateur.

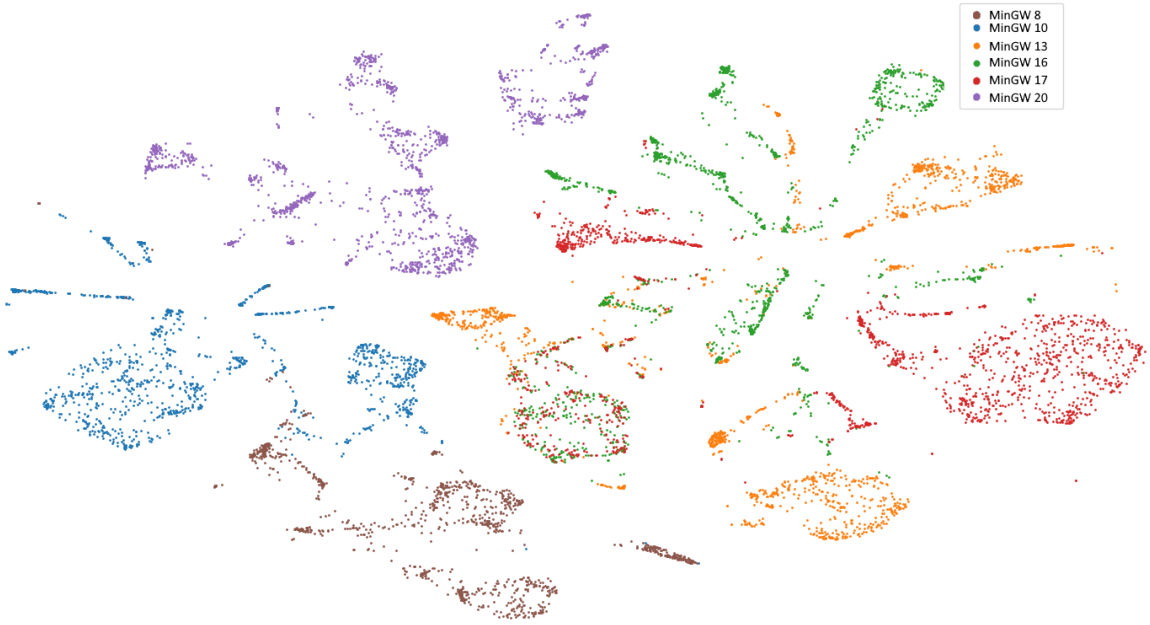


FIGURE 2.14 – Programmes compilés avec MinGW séparés en versions du compilateur.

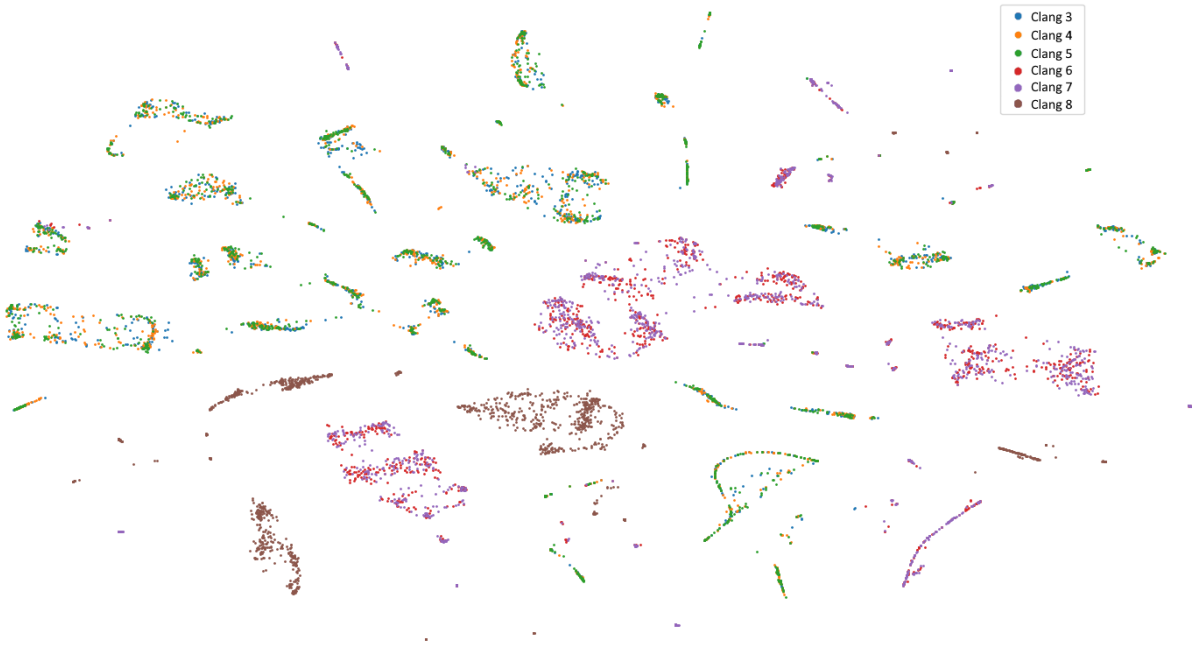


FIGURE 2.15 – Programmes compilés avec Clang séparés en versions du compilateur.

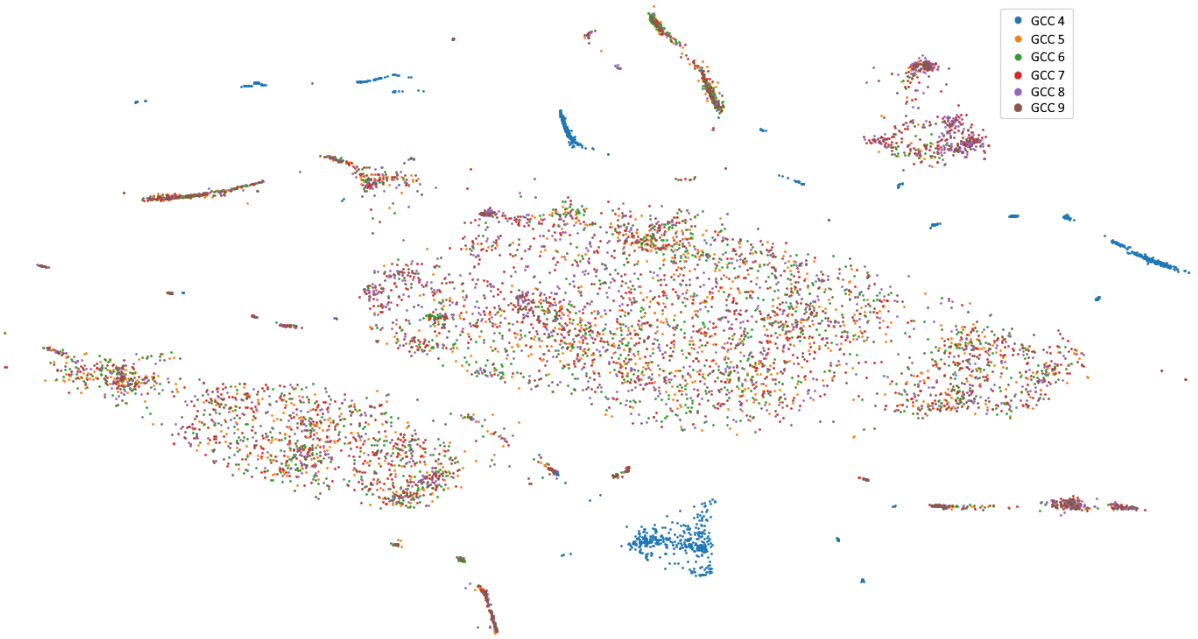


FIGURE 2.16 – Programmes compilés avec GCC séparés en versions du compilateur.

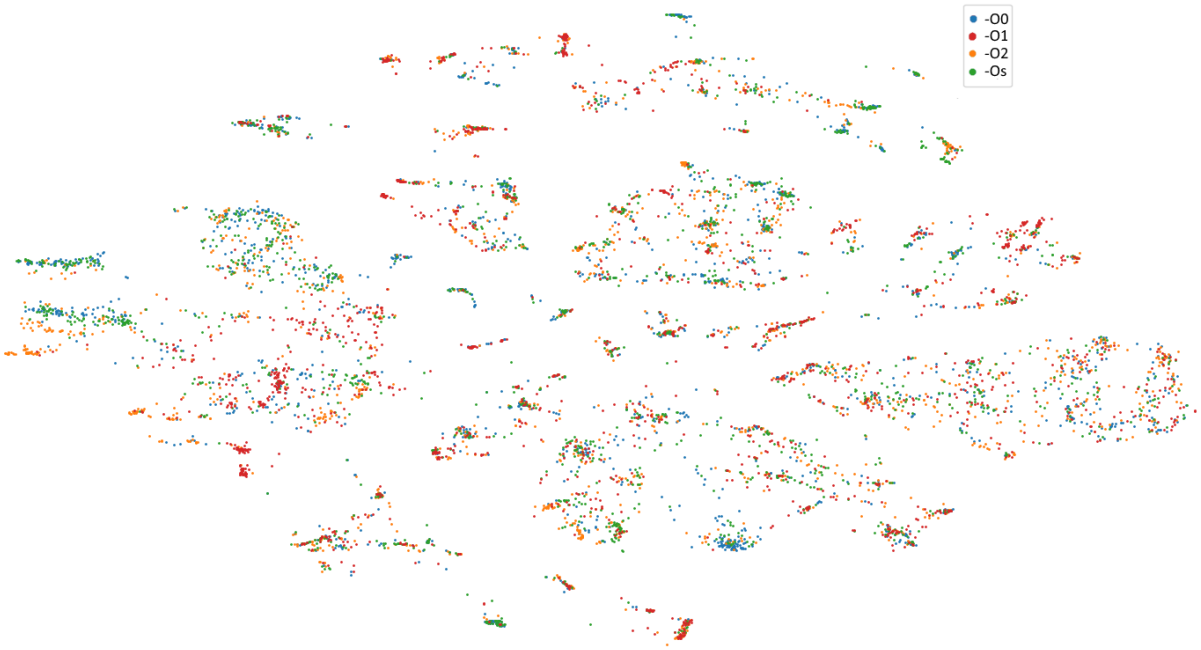


FIGURE 2.17 – Programmes compilés avec Visual Studio séparés en niveaux d'optimisation.

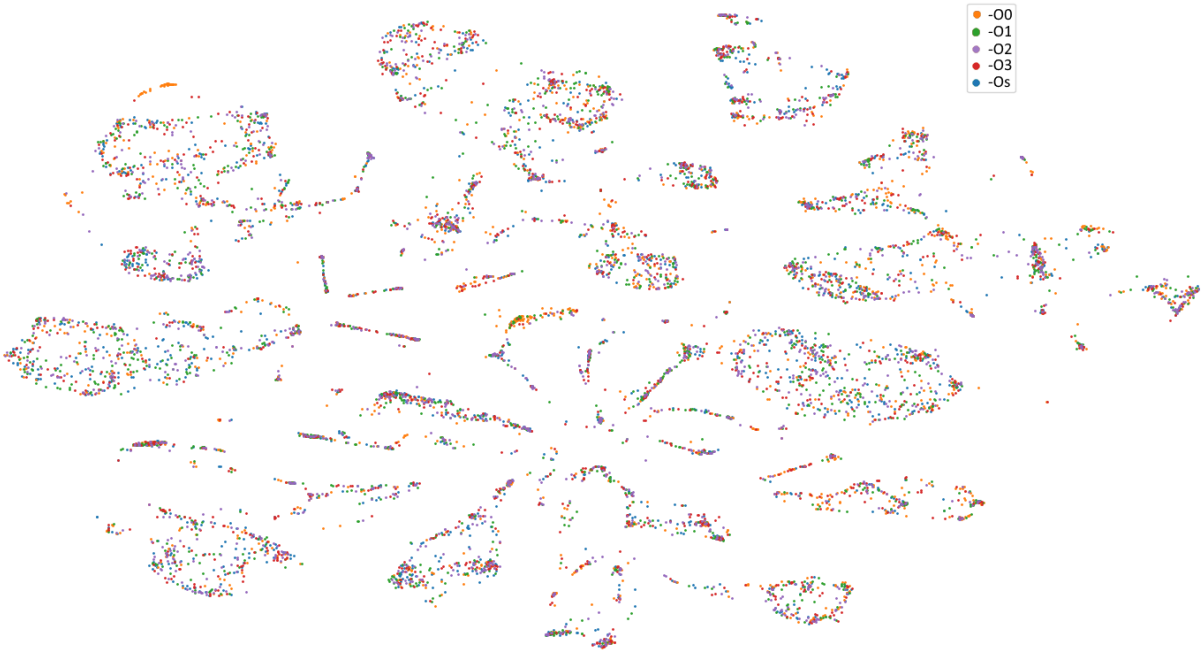


FIGURE 2.18 – Programmes compilés avec MinGW séparés en niveaux d'optimisation.

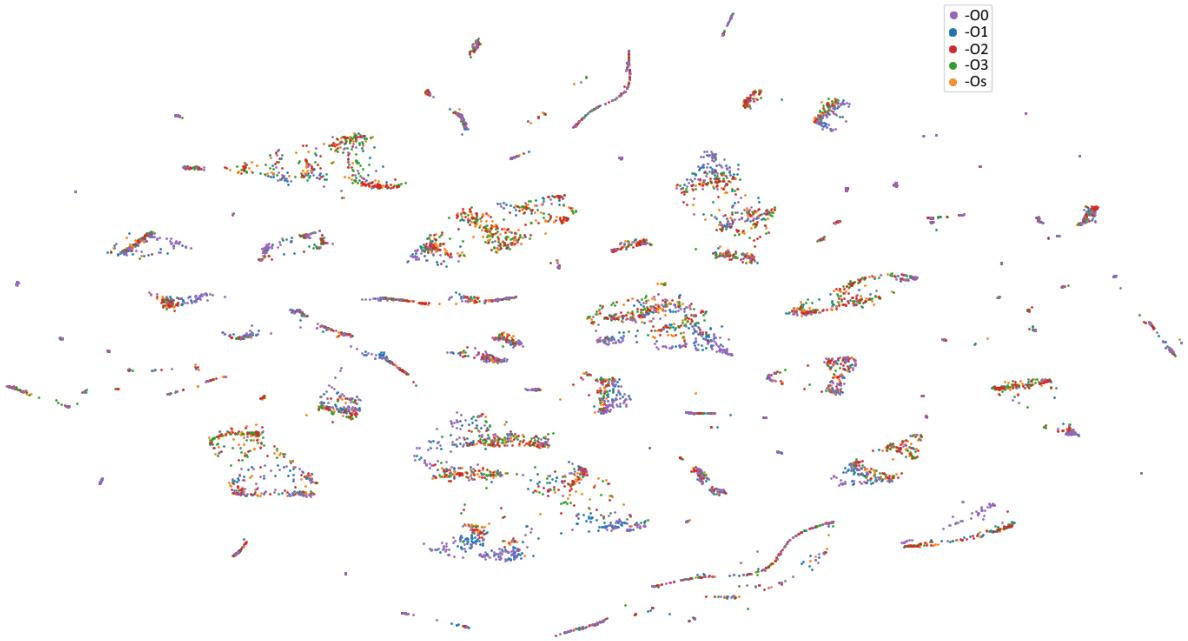


FIGURE 2.19 – Programmes compilés avec Clang séparés en niveaux d'optimisation.

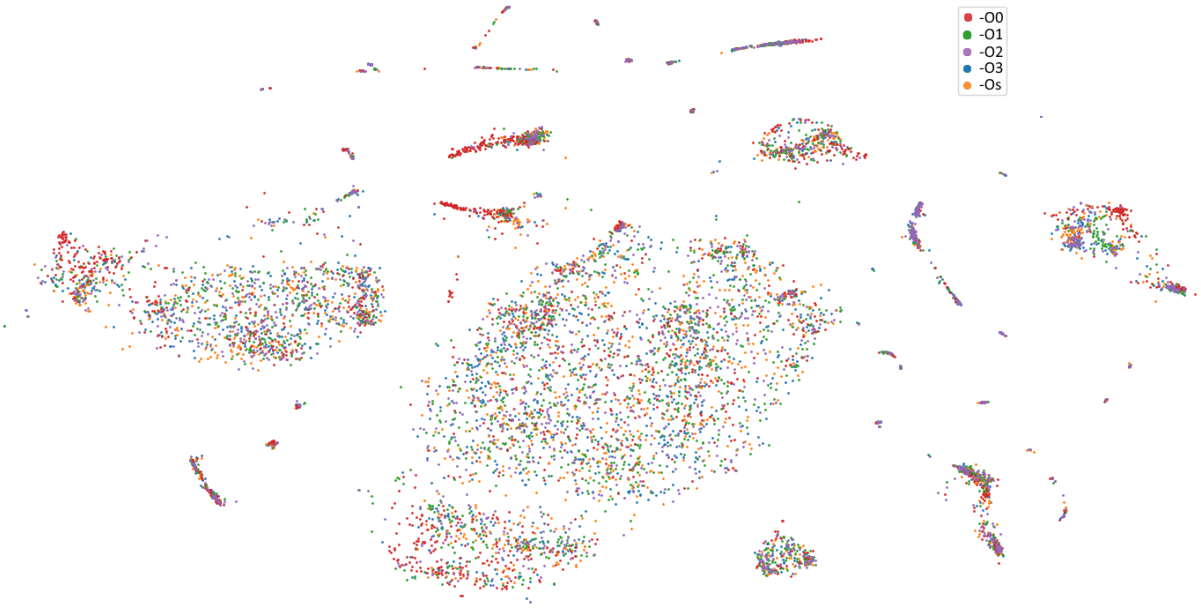


FIGURE 2.20 – Programmes compilés avec GCC séparés en niveaux d'optimisation.

2.7 Résultats

2.7.1 Impact du paramètre α sur la vitesse d'exécution

Le graphe de flot de contrôle contient des milliers de sommets. Pour pouvoir l'exploiter, nous le découpons en 100 graphes plus petits appelés des sites. Ces sites ont α sommets. Le paramètre α modifie le temps d'exécution du prétraitement ainsi que la phase d'apprentissage. Pour connaître la vitesse d'exécution, nous étudions les valeurs suivantes d'α : 4, 8, 12, 16, 20, 24, 28, 32, 48 et 64.

Comme illustré dans la Figure 2.21, avec ces valeurs d'α, la phase de découpage retient de 2 à 12 pour cent du CFG complet. C'est un bon intervalle puisque nous cherchons à conserver une petite partie du graphe.

Nous étudions les temps de prétraitement et le temps d'apprentissages d'une époque sur un échantillon aléatoire de 1000 programmes. À partir de cela, nous pouvons extrapoler le temps d'apprentissage complet suivant α. Nos expériences se déroulent sur un ordinateur équipé avec un Intel i7-8665U, 4 cœurs, et une fréquence de 2.11 GHz.

Conclusion L'extraction du CFG, qui ne dépend pas d'α, prend en moyenne $0,11$ secondes par programme. Le temps moyen de la phase de découpage va de $0,12$ seconde à $0,38$ seconde par programme en fonction d'α (Figure 2.22). Le temps moyen d'une phase d'apprentissage complète d'un programme va de $0,95$ seconde à $2,21$ secondes par programme (Tableau 2.2). Augmenter α augmente le volume de donnée à apprendre. Il faut donc adapter α à ses besoins. La phase d'extraction et de découpage peut être effectuée en parallèle aisément. De même, chaque classifieur local peut apprendre en parallèle.

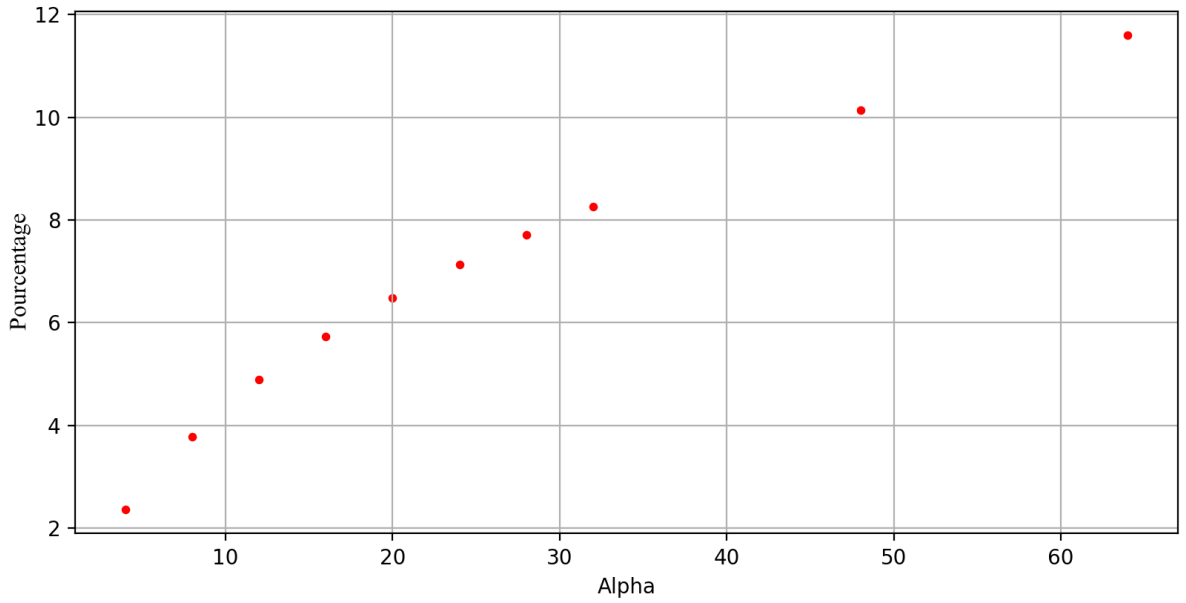


FIGURE 2.21 – Pourcentage du CFG original conservé par la phase de découpage (en Mo).

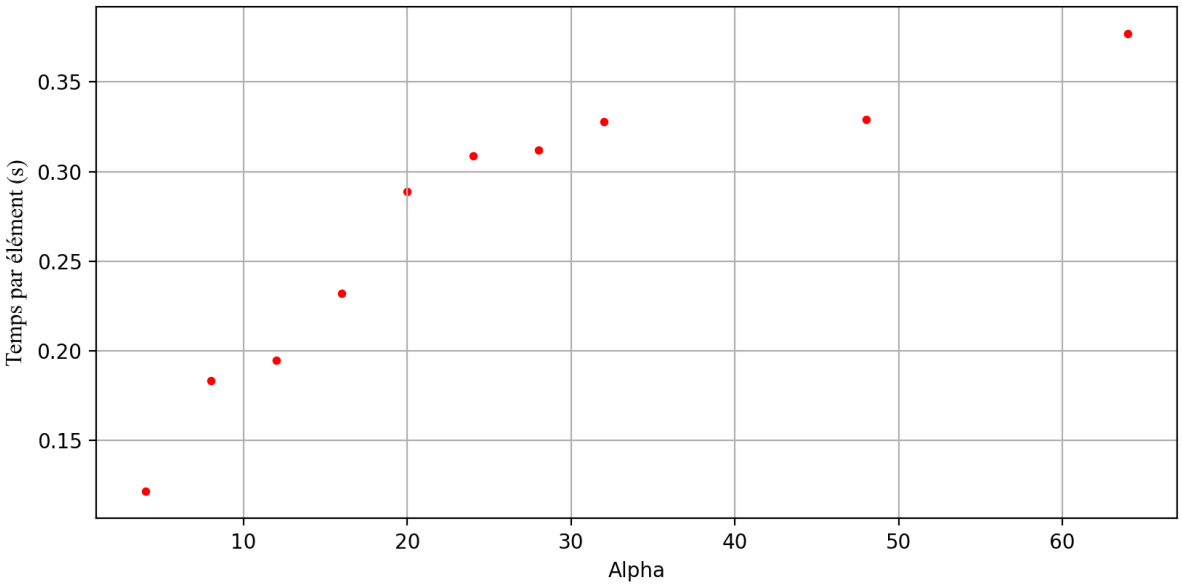


FIGURE 2.22 – Temps moyen de découpe d'un programme.

TABLE 2.2 – Temps moyens (s) de la phase d'apprentissage d'un programme.

α	Option	Version	Complet
4	0,46	0,58	0,95
8	0,59	0,74	1,21
12	0,73	0,80	1,39
16	0,71	0,85	1,42
20	0,85	0,97	1,67
24	0,79	0,90	1,54
28	0,84	0,93	1,61
32	0,85	1,02	1,71
48	0,98	1,08	1,88
64	1,13	1,25	2,20

2.7.2 Impact du paramètre α sur la précision

Nous devons étudier l'impact d'α sur la précision de notre processus. En effet, puisque α modifie le volume des données traitées, à première vue, cela doit affecter la capacité de prédiction de nos algorithmes. Pour mesurer la précision de nos hiérarchies, nous utilisons comme métrique le F1-Score. C'est une métrique simple qui n'est pas biaisée par des différences de présences entre certaines chaînes de compilations dans les données. Comme précédemment, nous étudions 4, 8, 12, 16, 20, 24, 28, 32, 48, et 64 comme valeurs d'α. Cependant, le résultat d'un apprentissage contient nécessairement une part de variance. Nous effectuons donc dix fois l'apprentissage pour chaque valeur d'α et nous considérons la moyenne des F1-Score. À chaque apprentissage, nous sélectionnons 10% de nos données comme jeu d'entraînement. Le meilleur modèle de chaque classifieur local va être sélectionné suivant ses performances par rapport à un sous jeu de validation de 10% de ce jeu d'entraînement. Enfin, à la suite d'un apprentissage, nous évaluons la précision par rapport à un échantillon aléatoire de 2000 programmes.

Nos résultats sont alors analysés par le biais de la méthode des moindres carrés ordinaire (OLS). Ce modèle statistique simple suppose que nos données suivent une loi linéaire à laquelle s'ajoute du bruit aléatoire distribué. Nous évaluons l'impact d'α sur la moyenne des F1-Score par la valeur r^2 de cette analyse. Nous donnons également la valeur-p, c.a.d. la probabilité d'obtenir une valeur au moins aussi grande que celle observée si α n'avait pas d'impact.

Concernant la prédiction du niveau d'optimisation, un modèle linéaire utilisant la valeur d'α ne permet pas de prédire les données (Figure 2.23). La probabilité qu'il n'y ait pas de relation est trop grande puisque la valeur p est de $0,537$. Même s'il y avait une relation, α expliquerait moins de 5% de la variance.

En revanche, concernant la prédiction de la version du compilateur, un modèle linéaire utilisant la valeur d'α est plus pertinent. Premièrement, la valeur p est de seulement $0,00991$. Cette probabilité pourrait être encore plus petite, puisque le test de Durbin-Watson a pour valeur $2,699$. En effet, à partir de deux, il est probable que des autocorrélations négatives ont artificiellement augmenté la valeur p. Ce serait lié à une supposition non respectée, mais nécessaire à l'utilisation de la méthode OLS.

Cependant, le résultat du test Omnibus est de $0,375$, ce qui est trop proche de zéro. La distribution du bruit n'est sans doute pas aléatoire et donc une autre supposition n'a pas été respectée. Le coefficient r^2 est de $0,585$, α expliquerait donc 58% de la variance dans la moyenne du F1-Score.

Conclusion Selon nos analyses, augmenter α accroit la capacité de prédiction de la version du compilateur d'un programme. Nous sélectionnons un α de 32 pour nos expériences ultérieures. Premièrement, cette valeur est la meilleure pour la prédiction de la version du compilateur, mais aussi pour la prédiction du niveau de compilation. Deuxièmement, c'est un compromis avec la vitesse d'exécution puisque nous extrayons seulement 8% du CFG originel (en Mo).

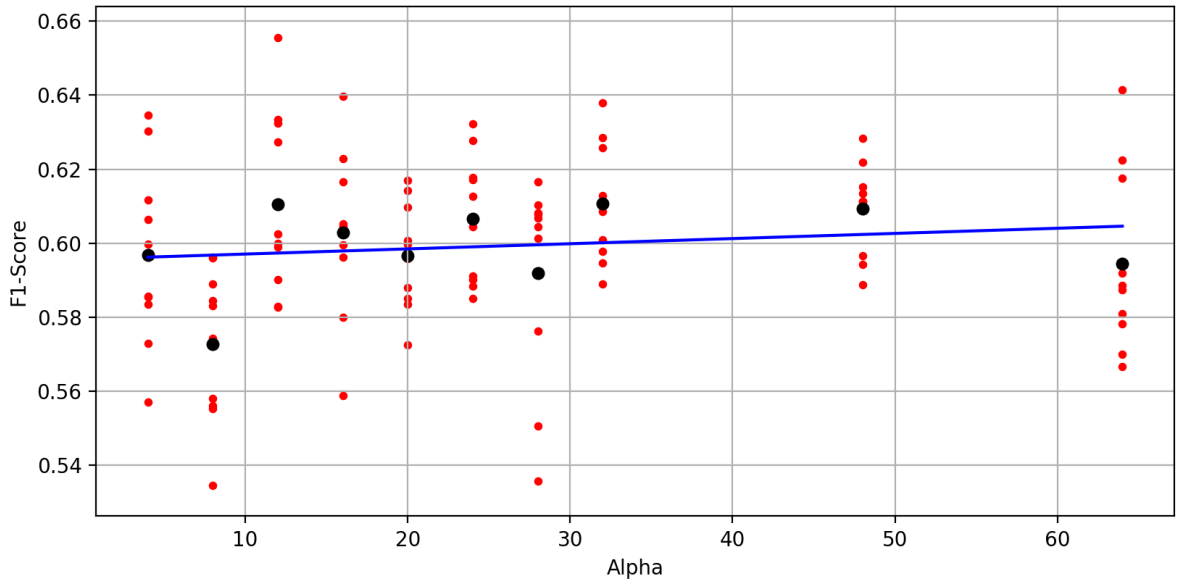


FIGURE 2.23 – F1-Score moyen selon α de la prédiction du niveau d'optimisation. Un point rouge est le résultat d'un apprentissage. Les points noirs sont les valeurs moyennes pour une valeur d'α. La droite en bleu est un modèle linéaire obtenu par la méthode OLS.

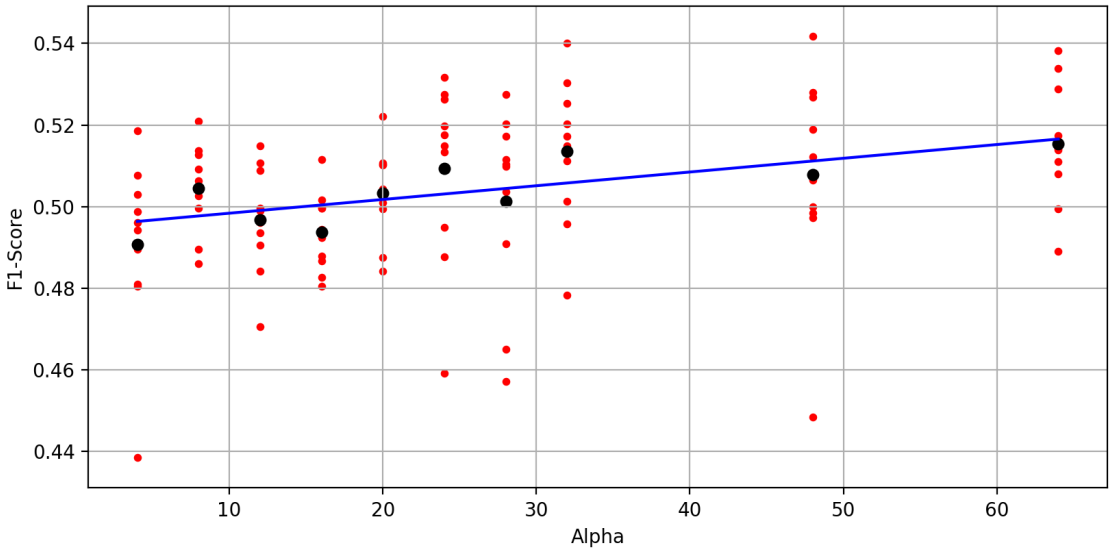


FIGURE 2.24 – F1-Score moyen selon α de la prédiction de la version du compilateur. Un point rouge est le résultat d'un apprentissage. Les points noirs sont les valeurs moyennes pour une valeur d'α. La droite en bleu est un modèle linéaire obtenu par la méthode OLS.

2.7.3 Précision

Nous évaluons la précision des deux hiérarchies en utilisant l'ensemble des données. Pour chaque classifieur local, un ensemble de 10% de l'ensemble d'entraînement est utilisé afin de sélectionner l'une des meilleures époques parmi les dix dernières. Un échantillon de 2200 programmes, équilibré au niveau des compilateurs, versions, et niveaux d'optimisation permet d'évaluer la précision de nos hiérarchies.

La Table 2.3 détaille les F1-Score de la prédiction des familles de compilateurs. Nous obtenons des F1-Score très élevés pour toutes les familles ainsi qu'une moyenne de $0,9950$.

La Table 2.4 détaille les F1-Score de la prédiction des niveaux d'optimisations. Nous obtenons une moyenne élevée de $0,7549$. Prédire le niveau d'optimisation -Os est le plus difficile, le score est de $0,63160$. En consultant la matrice de confusion (Figure 2.25), nous remarquons, chez les programmes compilés avec Clang, GCC, et MinGW, une confusion entre les programmes compilés et -Os et ceux compilés avec $O2/O3$. Par exemple, $53,33\%$ des programmes compilés avec Clang et l'option -Os sont prédits comme compilés avec Clang et l'option $O2/O3$. À l'inverse, 52% des programmes compilés avec Visual Studio et l'option -Os sont prédits comme compilés avec Visual Studio et l'option -O0.

TABLE 2.3 – F1-Score de la prédiction de la famille du compilateur.

Famille de compilateur	Précision	Rappel	F1-Score	Support
Clang	1	0,9933	0,9967	600
GCC	0,9967	1	0,9983	600
MinGW	0,9983	0,9917	0,9950	600
Visual Studio	0,9828	0,9975	0,9901	400
Moyenne	0,9944	0,9956	0,9950	2200

TABLE 2.4 – F1-Score de la prédiction du niveau d'optimisation.

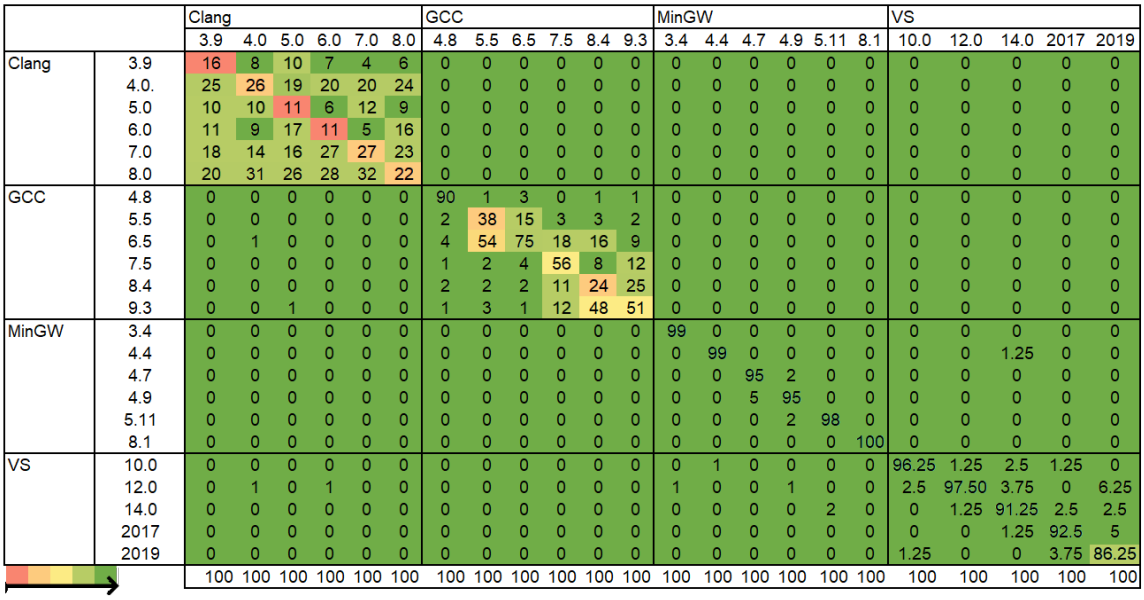
Niveau d'optimisation	Précision	Rappel	F1-Score	Support
-O0	0,7917	0,8261	0,8085	460
-O1	0,8289	0,7478	0,7863	460
$O2/O3$	0,7869	0,8329	0,8092	820
-Os	0,6316	0,6	0,6154	460
Moyenne	0,7598	0,7517	0,7549	2200

		Clang				GCC				VS				MinGW			
		O0	O1	O2/O3	Os	O0	O1	O2/O3	Os	O0	O1	O2	Os	O0	O1	O2/O3	Os
Clang	O0	95.83	5.83	0.83	1.67	0	0	0	0	0	0	0	0	0	0	0	0
	O1	1.67	55	4.58	2.5	0	0	0	0	0	0	0	0	0	0	0	0
	O2/O3	1.67	29.17	79.17	53.33	0	0	0	0	0	0	0	0	0	0	0	0
	Os	0.83	8.33	15	41.67	0	0	0	0	0	0	0	0	0	0	0	0
GCC	O0	0	0	0	0	98.33	6.67	0	0.83	0	0	0	0	0	0	0	0
	O1	0	0	0	0	1.67	92.5	2.5	0.83	0	0	0	0	0	0	0	0
	O2/O3	0	0	0.42	0	0	0.83	89.58	9.17	0	0	0	0	0	0	0	0
	Os	0	0	0	0.83	0	0	7.92	89.17	0	0	0	0	0	0	0	0
VS	O0	0	0	0	0	0	0	0	0	46	4	5	52	0.83	0	0.42	0.83
	O1	0	0.83	0	0	0	0	0	0	8	84	6	6	0	0	0	0.83
	O2	0	0	0	0	0	0	0	0	4	10	85	1	0	0.83	0	0
	Os	0	0.83	0	0	0	0	0	0	42	2	3	41	0	0	0	0
MinGW	O0	0	0	0	0	0	0	0	0	0	0	0	0	83.33	9.17	1.25	2.5
	O1	0	0	0	0	0	0	0	0	0	0	0	0	5.83	68.33	4.17	6.7
	O2/O3	0	0	0	0	0	0	0	0	0	0	1	0	5	16.67	79.58	25
	Os	0	0	0	0	0	0	0	0	0	0	0	0	5	5	14.58	64.17
		100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100

FIGURE 2.25 – Matrice de confusion de la prédiction de la famille du compilateur et du niveau d'optimisation. Sur la diagonal, la meilleure valeur est 100 et en dehors c'est 0.

La Table 2.5 présente les F1-Score de la prédiction des versions de compilateurs suivant les familles de compilateurs. Le F1-Score global est de $0,6475$. Nous remarquons que nos prédictions sont incorrectes pour la famille de compilateurs Clang, le F1-Score est de seulement $0,1856$. En revanche, le F1-Score pour la famille de compilateurs MinGW est de $0,9799$. Si on retire la famille Clang, nous obtenons un F1-Score de $0,8167$. De plus, si nous nous concentrons sur les programmes compilés pour le système d'exploitation Windows, nous obtenons un score très élevé de $0,9502$. En consultant la matrice de confusion (Figure 2.26), nous remarquons que parmi la famille de compilateurs Clang toutes les versions sont confondues entre elles. Ce n'est pas le cas pour la famille GCC chez qui une version est confondue avec les autres versions proches. Par exemple, les programmes compilés avec GCC 5.5 sont confondus dans 54% des cas avec des programmes compilés avec GCC 6.5. La Table 2.6 fournit plus de détails sur les F1-Score suivant les versions des compilateurs.

Conclusion Comme attendu, nous obtenons un F1-Score très élevé de $0,9944$ sur la prédiction de la famille du compilateur. La prédiction du niveau d'optimisation est plus difficile, particulièrement -Os. Les niveaux d'optimisations ne sont pas identiques suivant les compilateurs, comme l'atteste la matrice de confusion, ce qui justifie l'usage d'une hiérarchie de classifieurs locaux. Néanmoins, là où notre première analyse a indiqué que différencier les niveaux d'optimisations était difficile, nous atteignons le F1-Score élevé de $0,7598$. Enfin, si comme attendu la prédiction des versions de compilateur est facile pour les familles de compilateurs MinGW et Visual Studio, ce n'est pas le cas chez les familles de compilateurs Clang et GCC. Néanmoins, nous atteignons un F1-Score de $0,5496$ pour la famille de compilateurs GCC. Le F1-Score de la prédiction des versions de compilateur est de $0,6578$.



		Clang						GCC						MinGW						VS				
		3.9	4.0	5.0	6.0	7.0	8.0	4.8	5.5	6.5	7.5	8.4	9.3	3.4	4.4	4.7	4.9	5.11	8.1	10.0	12.0	14.0	2017	2019
Clang	3.9	16	8	10	7	4	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	4.0.	25	26	19	20	20	24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	5.0	10	10	11	6	12	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	6.0	11	9	17	11	5	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	7.0	18	14	16	27	27	23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	8.0	20	31	26	28	32	22	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
GCC	4.8	0	0	0	0	0	0	90	1	3	0	1	1	0	0	0	0	0	0	0	0	0	0	0
	5.5	0	0	0	0	0	0	2	38	15	3	3	2	0	0	0	0	0	0	0	0	0	0	0
	6.5	0	1	0	0	0	0	4	54	75	18	16	9	0	0	0	0	0	0	0	0	0	0	0
	7.5	0	0	0	0	0	0	1	2	4	56	8	12	0	0	0	0	0	0	0	0	0	0	0
	8.4	0	0	0	0	0	0	2	2	2	11	24	25	0	0	0	0	0	0	0	0	0	0	0
	9.3	0	0	1	0	0	0	1	3	1	12	48	51	0	0	0	0	0	0	0	0	0	0	0
MinGW	3.4	0	0	0	0	0	0	0	0	0	0	0	0	99	0	0	0	0	0	0	0	0	0	0
	4.4	0	0	0	0	0	0	0	0	0	0	0	0	0	99	0	0	0	0	0	0	1.25	0	0
	4.7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	95	2	0	0	0	0	0	0	0
	4.9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5	95	0	0	0	0	0	0	0
	5.11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	98	0	0	0	0	0	0
	8.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	100	0	0	0	0	0
VS	10.0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	96.25	1.25	2.5	1.25	0
	12.0	0	1	0	1	0	0	0	0	0	0	0	0	1	0	0	1	0	0	2.5	97.50	3.75	0	6.25
	14.0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	1.25	91.25	2.5	2.5
	2017	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1.25	92.5	5
	2019	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1.25	0	0	3.75	86.25
		100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100

FIGURE 2.26 – Matrice de confusion de la prédiction de la famille du compilateur et de sa version. Sur la diagonal, la meilleure valeur est 100 tandis qu'en dehors, elle est de 0.

TABLE 2.5 – F1-Score de la prédiction de la version du compilateur par famille.

Famille de compilateur	Précision	Rappel	F1-Score	Support
Clang	0,2019	0,1883	0,1856	600
GCC	0,5731	0,5567	0,5496	600
MinGW	0,9832	0,9767	0,9799	600
Visual Studio	0,9162	0,9275	0,9206	400
MinGW-Visual Studio	0,9497	0,9521	0,9502	1000
GCC-MinGW-Visual Studio	0,8242	0,8202	0,8167	1600
Toutes	0,6578	0,6508	0,6475	2200

TABLE 2.6 – F1-Score de la prédiction de la version d'un compilateur.

Version	Précision	Rappel	F1-Score	Support
Clang 3.9.1	0,3137	0,16	0,2119	100
Clang 4.0.1	0,194	0,26	0,2222	100
Clang 5.0.1	0,1897	0,11	0,1392	100
Clang 6.0.0	0,1594	0,11	0,1302	100
Clang 7.0.0	0,216	0,27	0,24	100
Clang 8.0.0	0,1384	0,22	0,1699	100
GCC 4.8.5	0,9375	0,90	0,9184	100
GCC 5.5.0	0,6032	0,38	0,4663	100
GCC 6.5.0	0,4237	0,75	0,5415	100
GCC 7.5.0	0,6747	0,56	0,612	100
GCC 8.4.0	0,3636	0,24	0,2892	100
GCC 9.3.0	0,4359	0,51	0,47	100
MinGW 3.4.5	1	0,99	0,995	100
MinGW 4.4.1	0,99	0,99	0,99	100
MinGW 4.7.1	0,9794	0,95	0,9645	100
MinGW 4.9.2	0,95	0,95	0,95	100
MinGW 5.11.0	0,98	0,98	0,98	100
MinGW 8.1.1	1	1	1	100
Visual Studio 10.0	0,939	0,9625	0,9506	80
Visual Studio 12.0	0,8478	0,975	0,907	80
Visual Studio 14.0	0,9125	0,9125	0,9125	80
Visual Studio 2017	0,9367	0,925	0,9308	80
Visual Studio 2019	0,9452	0,8625	0,902	80
Moyenne	0,6578	0,6508	0,6475	2200

2.7.4 Comparaison avec Massarelli et al. [105]

Nous nous comparons au travail de Massarelli et al. [105]. Comme dit précédemment, c'est l'un des travaux les plus proches. En effet, ils utilisent des convolutions sur les graphes de flot de contrôle. De plus, un dépôt contient l'infrastructure logicielle. Ce n'est pas le cas pour les autres travaux [32, 79, 121, 126, 159], ce qui rend la comparaison difficile. Dans le cas de Chen et al. [32], il est prétendu que l'infrastructure logicielle est disponible, mais nous ne l'avons pas trouvée.

Massarelli et al. [105] utilisent radare2 [142] pour extraire le CFG de chaque fonction binaire. Ils proposent une méthode de vectorisation des instructions nommée I2V et utilisent un réseau de neurones récurrent. Ils ont entraîné I2V sur un corpus large de programmes venant du système d'exploitation Linux. Cependant, une projection aléatoire produit selon eux des résultats proches d'I2V. Nous pouvons réutiliser un modèle préentraîné de leur méthode de vectorisation d'instructions. Bien que leur article ne démontre des résultats que sur la prédiction de la famille du compilateur, la prédiction des niveaux d'optimisation et des versions de compilateur est prévue par leur infrastructure logicielle.

Comme nous allons le démontrer, cette autre infrastructure est bien plus lente. Nous devons sélectionner un petit échantillon de notre jeu de données complet pour la comparaison. Nous sélectionnons 2 843 programmes de notre jeu de données dont nous extrayons les graphes de flot de contrôle des fonctions binaires. Dans l'infrastructure de Massarelli et al., 227 programmes sont là pour sélectionner les modèles, 601 constituent le jeu d'évaluation et le reste est utilisé pour l'entraînement. De notre côté, nous entraînons nos hiérarchies sur les mêmes 2 843 programmes avec un jeu d'évaluation de 601 programmes. Nous répétons le processus dix fois pour obtenir une bonne estimation de notre précision.

Afin de comparer une approche par fonction binaire et une approche par programme, nous distinguons les deux sorties :

- **Fonction** Nous évaluons la capacité de prédiction sur les fonctions binaires de l'infrastructure de Massarelli et al., noté MA. Nous transformons la sortie de notre infrastructure. La prédiction de chaque fonction contenue dans un programme est celle prédite pour le programme entier. Nous notons cette infrastructure SNN-F ;
- **Programme** Nous avons déjà évalué notre infrastructure sur le jeu de donnée complet, et notons le résultat comme SNN-Complet. Nous transformons la sortie de MA. La prédiction d'un programme entier est la prédiction majoritaire parmi les fonctions binaires du programme. Nous notons cette infrastructure MA-B.

Le prétraitement de Massarelli et al. est complexe à cause du désassemblage de radare2. En moyenne un programme prend 665 secondes à être désassemblé, et une minorité prend plusieurs heures. Ce prétraitement peut être effectué en parallèle, mais il prend en tout 525 heures. La machine est équipée de deux cœurs Intel Xeon Silver 4110. En tout, près de 825 000 fonctions binaires sont extraites.

Chaque époque de l'apprentissage prend 2 200 secondes. Ce qui revient à un temps de 116 secondes par programme. L'apprentissage est effectué avec deux cœurs Intel Xeon Gold 5218R.

TABLE 2.7 – F1-Score des différentes infrasctuctures.

Infrastructure / Tâche	Famille	Niveau d'optimisation	Version
MA-B	0,91	0,42	0,32
SNN	0,92 [±0, 03]	0,58 [±0, 03]	0,45 [±0, 02]
SNN-Complet	**0,99**	**0,75**	**0,65**
MA	**0,90**	0,36	0,36
SNN-F	0,87 [±0, 07]	**0,60** [±0.05]	**0,42** [±0, 02]

Notre approche, avec un modeste processeur Intel i7-8665U, a un temps d'apprentissage de $1,71$ seconde par programmes. Nous présentons les résultats dans la Table 2.7. Étant donné le faible échantillon, nous ne nous intéressons qu'aux F1-Score moyens et nous utilisons la moyenne ainsi que la déviation standard de dix lancées de l'expérience.

Concernant les prédictions pour des programmes entiers, notre infrastructure SNN est aussi bonne que MA-B pour prédire la famille du compilateur. Mais, nous dépassons MA-B de $0,16$ sur la prédiction des niveaux d'optimisation et de $0,13$ sur la prédiction des versions de compilateurs. Si on considère le jeu de données complet, SNN-Complet, nous dépassons de $0,08$ sur la prédiction de la famille du compilateur, $0,33$ sur la prédiction des niveaux d'optimisation et $0,33$ sur la prédiction des versions de compilateurs. Néanmoins, MA-B a eu moins de données à disposition pour s'entraîner que SNN-Complet.

Concernant les prédictions pour des fonctions binaires, nous comparons MA et SNN-F. Là encore, nous sommes au même niveau sur la prédiction de la famille du compilateur. Nous dépassons MA de $0,24$ sur la prédiction des niveaux d'optimisation et de $0,06$ sur la prédiction des versions de compilateur.

Conclusion Notre proposition est 68 fois plus rapide durant l'apprentissage, et 1 300 fois plus rapide durant le prétraitement. De plus, nos hiérarchies permettent plus facilement d'utiliser le parallélisme. Cette vitesse d'exécution permet d'améliorer nos précisions en intégrant plus de programmes. Même sans tirer parti de cet avantage, nous sommes plus précis que l'infrastructure MA pour prédire les niveaux d'optimisation et les versions de compilateurs.

2.7.5 Comparaison avec Rosenblum et al. [124]

Malgré les difficultés dues à l'indisponibilité de leurs jeux de données ou de leur infrastructure , nous fournissons des éléments de comparaison avec les travaux de Rosenblum et al. [124].

Ils rapportent une précision de $0,999$ pour la prédiction de la famille du compilateur, $0,999$ pour la prédiction du niveau d'optimisation, et de $0,918$ pour la version du compilateur.

Néanmoins, leur jeu de donnée contient 2 686 programmes compilés depuis *seulement 175 codes sources*. Nous considérons de notre côté 36 272 codes sources (la validité interne [27] est

préservée par l'assignation d'un code source à une chaîne de compilation par le biais de l'aléatoire). Sont considérées seulement trois familles de compilateurs (Visual Studio, Intel Compiler, et GCC), et 9 versions de compilateurs en tout. De plus, le niveau d'optimisation est considéré soit comme faible (p. ex. -O0), soit comme élevé (p. ex. -O3). Au total, ils considèrent 18 chaînes de compilation contre 92.

Pour observer l'impact de la restriction des chaînes de compilation, nous choisissons de considérer seulement les versions de compilateurs Visual Studio 10.0, Visual Studio 12.0, Visual Studio 2017, GCC 4.8, GCC 5.5 et GCC 7.5 et de nous restreindre à prédire un niveau d'optimisation faible ou élevé. Nous constatons que le F1-Score de la prédiction de la famille du compilateur est 1, celui de la prédiction du niveau d'optimisation est de $0,97$ et enfin celui de la prédiction de la version du compilateur est de $0,89$.

Comme attendu, la portée du jeu de donnée a un effet fort sur la qualité des prédictions.

2.8 Conclusion

Nous avons considéré le problème de la prédiction de la chaîne de compilation d'un programme. Notre hypothèse de départ est qu'un programme n'est pas une donnée sans structure et que la sémantique est importante. De plus, comme les bibliothèques sont plus souvent liées dynamiquement, les programmes sont généralement homogènes en termes de chaîne de compilation.

Dans ce travail, nous explorons la possibilité (i) d'extraire des caractéristiques sémantiques sous forme de graphes (ii) d'appliquer des réseaux de neurones sur des petits graphes appelés sites, et (iii) d'utiliser des hiérarchies spécifiques à un domaine. Nous démontrons que l'identification de la chaîne de compilation d'un programme complet est possible, avec une haute précision et rapidement.

Ce travail ouvre plusieurs questions. La combinaison de la phase de réduction suivie d'un découpage fournit des caractéristiques simples et pertinentes. Cela dit, on pourrait imaginer un prétraitement qui conserverait plus d'informations. Les couches de regroupement des GNN jouent également un rôle important. On pourrait observer quelles caractéristiques sont utiles afin d'améliorer le regroupement. Cette question se recoupe avec celle de la sémantique. Notre jeu de donnée est composé de petits programmes (la plupart pèsent moins de 30 Ko). Ce serait intéressant d'utiliser un jeu de données plus large. De plus, on pourrait considérer des programmes offusqués ou des programmes malveillants. Enfin, le CFG ne contient qu'une sémantique peu profonde. Une piste intéressante est d'extraire automatiquement et de tirer profit de caractéristiques sémantiques plus riches sans trop perdre en vitesse d'exécution.

Chapitre 3

La recherche de clones

Le Chapitre 3 introduit la tâche principale de cette thèse, la recherche de clones de programmes et notre proposition PSS basée sur les méthodes spectrales. Nous fournissons une description complète du problème de la recherche de clones ainsi qu'un exemple motivant la nécessité d'une nouvelle approche par rapport à la recherche de clones de fonctions. Nous présentons ce qu'est la similarité entre des programmes. Nous détaillons les difficultés que pose ce problème, qui sont liées à l'impact des chaînes de compilation ou à la taille de jeux de données, et l'état de l'art. Nous esquissons nos contributions, que ce soit nos méthodes spectrales, notre cadre d'évaluation dans les Chapitres 4 et 5 ainsi que nos résultats. Enfin, nous présentons ce qu'est l'analyse spectrale sur les graphes, ainsi que notre contribution PSS, une méthode spectrale, ses avantages et leurs liens avec le problème de la distance d'édition des graphes.

3.1 Introduction

La recherche de similarité de code binaire consiste à identifier des similarités ou des différences [61] entre des morceaux de code (p. ex., des blocs de base, des fonctions binaires ou des programmes entiers). Nous nous concentrons sur la similarité entre programmes (dite *similarité de programme* dans la suite), c.a.d. que nous calculons un indice de similarité capable de dire à quel degré deux programmes sont similaires.

Recherche de clones de programme Une *requête* est composée d'un *programme cible* et d'un dépôt, la *recherche de clones de programmes* classe les programmes du dépôt par leur similarité avec le programme cible. La recherche est réussie si le programme le plus similaire est un *clone* du programme cible. Ces clones peuvent avoir été compilés à partir du même code source avec une chaîne de compilation différente, ou produits à l'aide d'une version légèrement différente du code source. Le dépôt est une base de données contenant l'information suffisante à la procédure de recherche de clones. En pratique, un dépôt est relativement exhaustif suivant le domaine d'application (logiciels embarqués, plagiat, programmes malveillants, etc.).

Applications La recherche de similarités entre des programmes est nécessaire lorsque le code source original est indisponible, ce qui se produit avec les programmes prêts à l'emploi (COTS en anglais), les systèmes hérités, les programmes intégrés ou les logiciels malveillants. Détecter les clones de logiciels malveillants et les classer sont des problèmes majeurs [7, 41, 109, 152]. Selon un rapport de l'Agence de cybersécurité et de sécurité des infrastructures (CISA) [10], la plupart des familles de logiciels malveillants sont en circulation depuis plus de cinq ans, mais de nouveaux variants rendent difficiles les détections. Une autre application est l'identification de bibliothèques [6, 43, 64, 72, 140, 141], qui est à la fois un problème d'ingénierie logicielle et un problème de cybersécurité en raison des vulnérabilités à l'intérieur de bibliothèques.

Des fonctions aux programmes Les cinq dernières années ont vu un fort intérêt pour des approches basées sur l'apprentissage automatique appliqué à la similarité de fonctions binaires [42,101,106,154,158]. Le problème de l'identification des bibliothèques, tout en étant entre les programmes et les fonctions en termes de taille, est beaucoup plus proche du cas des clones de programmes par sa nature - les bibliothèques ne sont pas des collections arbitraires de fonctions et nécessitent une analyse interprocédurale. Dans tous ces cas, nous considérons l'approche au niveau de la fonction comme étant seulement une approximation du problème qui est, par nature, au niveau du code binaire entier. De plus, il est nécessaire pour l'analyse de correctifs et de logiciel intégré [155], ou la détection du plagiat [43,64,111], de considérer une vue plus globale du code.

Exemple fictif Supposons que nous ayons un dépôt de programmes dans le domaine du logiciel embarqué. Ce dépôt contiendrait des milliers de programmes développés pour différents dispositifs électroniques tels que des téléphones, des tablettes, des systèmes de navigation, etc. Nous pourrions avoir pour programme cible une bibliothèque de traitement d'image pour un téléphone. Trouver un clone de ce programme dans le dépôt pourrait alors nous permettre de découvrir des bibliothèques qui ont été développées pour d'autres dispositifs. Ce serait utile pour la détection de plagiat ou pour estimer l'impact d'une vulnérabilité présente dans toutes les versions d'une bibliothèque.

3.2 Difficultés

La recherche de clones de programmes présente de nombreux défis. Comme déjà mentionné, cela nécessite la comparaison de programmes entiers, c.a.d. des objets beaucoup plus grands que les fonctions, et les calculs de similarité doivent être adaptés aux tailles typiques des programmes.

De plus, nous ne considérons pas deux programmes pris isolément, mais un programme cible et un dépôt (possiblement volumineux) de programmes, d'où la nécessité de calculs de similarité très efficaces qui seront itérés sur tous les programmes du dépôt.

10. `https://www.cisa.gov/uscert/ncas/alerts/aa22-216a`

Par ailleurs, le dépôt peut contenir des programmes similaires mais légèrement différents, en raison de variations de la chaîne de compilation ou de la version du code. Enfin, la technique doit fonctionner également sur les codes binaires dépouillés (dont les symboles ont été retirés), gérer le cas où les noms de fonctions externes sont indisponibles (p. ex. le code intégré des objets connectés) et gérer les faibles offuscations (telles que l'ajout de code mort ou la dissimulation d'identificateurs).

Toutes ces contraintes ne s'adaptent pas bien aux travaux antérieurs sur la similarité, car l'état de l'art se concentre de plus en plus sur la *similarité de fonctions binaires*. Selon Haq et Caballero [61], depuis 2014, parmi 40 approches de similarité de code binaire, seules 7 approches acceptent des programmes en entrée. L'adaptation est incertaine vers le cas du niveau du programme.

Par exemple, nous constatons dans nos expériences que SMIT [69] prend plus de 43 heures pour calculer un indice de similarité entre la bibliothèque principale de Geany [11] et la commande cp, tandis que DeepBinDiff [44] prend 10 minutes sur de petits programmes du paquet Coreutils.

En vue de résoudre le problème de la recherche de clones de programmes, il y a un besoin fort d'une technique de similarité qui soit à la fois *précise*, *robuste* face à de légères variations et assez *rapide* pour pouvoir fonctionner sur de larges dépôts de code contenant des dizaines de milliers de programmes dont des bibliothèques.

3.3 Contributions

Nous explorons l'application de l'analyse spectrale des graphes [34] au problème de recherche de clones de programmes. Cette idée semble être un très bon point de départ, car, l'analyse spectrale est à la fois rapide et comparable à la distance d'édition de graphes [129] en termes de précision. Cependant, les programmes ne sont pas des graphes standard. D'une part, les programmes vus comme des graphes peuvent être très grands, surtout au niveau binaire. D'autre part, les programmes sont fortement structurés en raison des appels de fonctions. Nous profitons de ces spécificités et proposons **Program Spectral Similarity (PSS)**, la première analyse spectrale adaptée à la similarité des *programmes*. Cette analyse extrait des *valeurs propres* liées à la fois aux graphes d'appels de fonctions et aux graphes de flot de contrôle, et profite d'une étape de *prétraitement* (effectuée une fois pour l'ensemble des programmes du dépôt) pour réaliser des calculs de similarité en temps *linéaire par rapport au nombre de fonctions du programme* (effectué pour chaque programme dans le dépôt), la plupart des travaux antérieurs étant au moins quadratiques.

Nous montrons expérimentalement que PSS surpasse l'état de l'art et est résistante aux variations de code ainsi qu'à certaines offuscations. De plus, PSS ne s'appuie pas sur les *identificateurs littéraux* (p. ex., noms de fonctions, chaînes de caractères constantes), ce qui le rend robuste face à une gamme d'offuscations. Dans nos expériences, une recherche de clones de programmes avec

11. Un éditeur de texte, voir le site officiel à l'adresse : `https://www.geany.org/`

la version optimisée de PSS prend en moyenne moins de 3s, avec des temps de 0,3 et 0,4 secondes pour les jeux de données BinKit (97 760 programmes) et IoT (19 959 programmes) respectivement. En comparaison, la méthode par vectorisation de fonctions Gemini [154] nécessite environ 2 minutes par recherche de clones sur les petits dépôts de programmes de notre jeu de données Basique.

Nous mettons en place un cadre d'évaluation robuste et exhaustif (15 autres outils et 3 points de comparaison) pour comparer systématiquement PSS avec les méthodes de l'état de l'art, couvrant les méthodes basées sur les chaînes de caractères [140, 141], la distance d'édition de graphes [54, 69], les N-grammes [68], la vectorisation de fonctions [42, 101, 106, 154], les méthodes spectrales standard [54] et les algorithmes de couplage [7, 152]. Nos expériences couvrent notre propre jeu de données de paquets libres, ainsi que les paquets Coreutils, Diffutils, Findutils et Binutils avec deux dimensions de variations (niveaux d'optimisation et versions de code) pour un total de 950 binaires différents. De plus, nous considérons une partie du jeu de données BinKit [85] (96K binaires), qui couvre quatre niveaux d'optimisation, 9 compilateurs, 8 architectures et 4 offuscations. Enfin, nous rassemblons 19 959 programmes malveillants ciblant les objets connectés et 84 992 programmes bienveillants créés pour les systèmes d'exploitation de la famille Windows.

En résumé, nous rapportons quatre contributions principales portant la recherche de clones.

Première contribution Une nouvelle technique nommée PSS, et sa version optimisée PSSO, pour la similarité de code binaire (Section 3.7), adaptée à la recherche de clones de *programmes* sur de *grands* dépôts. PSS est la première technique spectrale adaptée à la similarité des programmes. En particulier, PSS profite d'une étape de prétraitement pour effectuer des calculs ultérieurs de similarité en temps linéaire par rapport au nombre de fonctions du programme, ce qui en fait un excellent outil pour la recherche de clones de programmes sur de grands dépôts.

Seconde contribution Un cadre d'évaluation exhaustif pour la recherche de clones de programmes (Section 4.3), comprenant (1) 97 760 programmes de BinKit [85], 19 959 programmes malveillants ciblant les objets connectés, 84 992 programmes Windows et un plus petit jeu de données Linux de 950 programmes, ainsi que (2) trois méthodes basiques et 15 méthodes de l'état de l'art - 11 d'entre elles étant réimplémentées. *Le cadre d'évaluation complet est disponible en ligne*, ce qui est rare dans ce domaine [104].

Troisième contribution Des preuves expérimentales (Sections 4.4 et 4.5) que PSS atteint un compromis idéal en termes de vitesse, de précision et de robustesse, ce qui la rend parfaite pour la recherche de clones de programmes, alors que les travaux antérieurs dans le domaine sont plus spécialisés dans l'évaluation des similarités à l'échelle des fonctions. En particulier, PSS semble bien se comporter et conserver une bonne précision dans des scénarios de recherche de clones exigeants (changement de compilateur, ou d'architecture matérielle).

Quatrième contribution Enfin, en tant que résultats secondaires, nous montrons que de nombreux travaux antérieurs ciblant la recherche de clones de fonctions ne peuvent pas faire face à la recherche de clones de programmes en raison de problèmes de temps d'exécution, et nous proposons deux nouvelles méthodes simples basées sur des identificateurs littéraux (chaînes de caractères constantes ou noms des fonctions externes). Malgré leur simplicité, ces deux méthodes semblent bien se comporter lorsque ces identificateurs sont disponibles.

Outre le développement d'une méthode novatrice et efficace pour la recherche de clones de programmes, nos résultats apportent une perspective nouvelle sur les travaux antérieurs portant sur la similarité de code. Tout d'abord, nous mettons en avant le cas de la recherche de clones de programmes et montrons qu'il se comporte différemment du cas bien étudié de la recherche de clones de fonctions. Des méthodes dédiées sont donc nécessaires. Deuxièmement, nous sommes les premiers à repérer la séparation entre les techniques utilisant des identificateurs littéraux et celles qui n'en utilisent pas. Nous identifions deux méthodes simples (FunctionSet et StringSet) comme porteuses lorsque ces littéraux sont disponibles. Troisièmement, nous montrons le potentiel des méthodes spectrales dédiées à la recherche de clones de programmes. Dans l'ensemble, nous croyons que ces résultats ouvrent la voie à de nouvelles directions de recherche dans le domaine.

Des artéfacts de nos recherches sont disponibles sur Zenodo [18].

3.4 Formalisation du problème

Étant donné un programme inconnu *cible* P et un *dépôt* de programmes D, l'objectif est d'identifier un *clone* de P dans D. Notez que tout au long nous supposons qu'il n'y a pas de copie exacte de la cible dans le dépôt D.

Un clone d'un programme P est défini comme suit :

- Un programme Q compilé à partir du même code source S que P, mais avec une chaîne de compilation différente, est un clone de P. Par exemple, P a été compilé avec GCC v9.1 en utilisant le niveau d'optimisation -O0 à partir du code source S, et Q a également été construit à partir de S en utilisant le même compilateur, mais un autre niveau d'optimisation, tel que -O3 ;
- Un programme Q compilé à partir d'une autre version du code source de P est un clone de P. Par exemple, les deux instances de l'application git compilées à partir de deux versions du code source, disons v2.35.2 et v2.37.1, sont des clones.

Dans le dernier cas, nous devons être un peu prudents. En effet, nous ne pouvons considérer que des versions incrémentales d'une application ou d'une bibliothèque, pas des révisions majeures qui modifient complètement le code source.

3.4.1 Difficultés

Les procédures de recherche de clones doivent prendre en compte les trois critères ci-dessous :

- L'efficacité par rapport à la taille du programme cible et la taille du dépôt ;

- La robustesse non seulement par rapport aux chaînes de compilation, mais également aux variations provenant de légères différences de code source ;
- La capacité à gérer les programmes dépouillés de symboles. De plus, certains identificateurs comme des noms de fonctions externes ne sont pas nécessairement disponibles lors de la manipulation de micrologiciels, d'offuscations légères ou encore de charges virales extraites de programmes malveillants offusqués [33].

3.4.2 Des fonctions aux programmes

Comme nous l'avons dit précédemment, la principale différence entre la recherche de clones de programmes et la recherche de clones de fonctions est la taille des codes binaires, qui est beaucoup plus grande dans le cas des programmes. Toutefois, adapter la recherche de clones de fonctions pour détecter des clones de programmes est attrayant.

Pour obtenir un indice de similarité entre deux programmes à partir de méthodes par vectorisation de fonctions, nous devons trouver comparer des ensembles de fonctions vectorisées. Soit *embeds* un outil de vectorisation de fonctions qui pour un programme P renvoie l'ensemble des vecteurs de fonctions. Notre solution est de réaliser un couplage entre les deux ensembles $embeds(P)$ et $embeds(P')$. Un tel couplage pourrait être une instance du problème d'affectation dans lequel le couplage d'un vecteur x de P avec un vecteur y de P' a pour coût $\|x - y\|_2$.

Avec l'algorithme hongrois [92], la résolution s'effectue en temps $O(n^3)$ où n est le nombre de fonctions. Puisque c'est trop important, nous relâchons la contrainte du couplage de façon à ce qu'un vecteur de P puisse être assigné à plusieurs vecteurs de P'. De cette façon, la complexité en temps est réduite à $O(n^2)$.

Nous définissons la métrique de similarité F qui munit d'un outil de vectorisation de fonctions *embeds* produit un indice de similarité entre deux programmes :

$$F(P, P') := - \sum_{x \in embeds(P)} \min_{y \in embeds(P')} \|x - y\|_2 \tag{3.1}$$

3.4.3 Architecture des procédures de recherche

À haut niveau, toutes les procédures de recherche de clones de programmes fonctionnent de la même manière. Le dépôt est déjà constitué, et le processus de requête est divisé en trois étapes :

1. **Prétraitement de la requête.** Lors de la requête, nous recevons le programme cible P. Nous pouvons effectuer un prétraitement à cette étape, c'est dire extraire des caractéristiques pertinentes pour le reste de la procédure ;
2. **Calcul de similarités.** Pour chaque programme $Q \in D$, nous effectuons un calcul de similarité avec une métrique de similarité M sur (P, Q) - en profitant éventuellement du prétraitement - et enregistrons l'indice de similarité calculé $M(P, Q)$;
3. **Décision.** Le programme Q_m ayant l'indice de similarité le plus élevé est considéré comme le plus similaire. La recherche de clones est un succès si Q_m est un clone de P, sinon c'est un échec.

La Figure 3.1 illustre les trois étapes d'une procédure de recherche de clones.

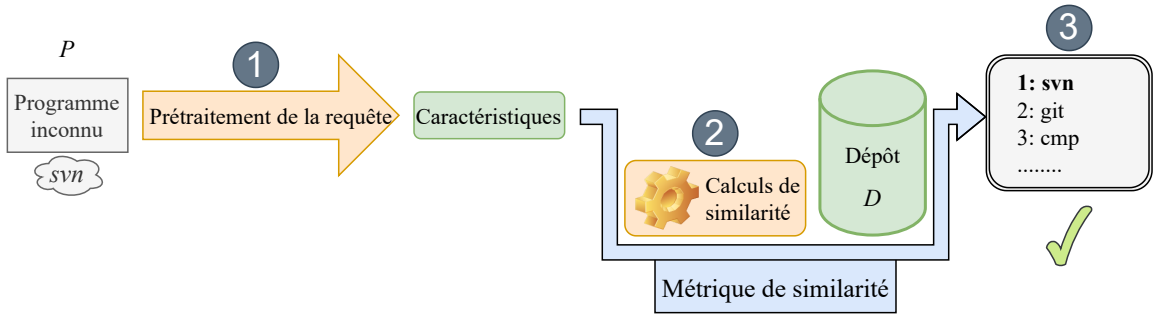


FIGURE 3.1 – Architecture d'une procédure de recherche de clones.

3.5 Un exemple

Nous considérons un dépôt contenant 1420 bibliothèques obtenues à partir de la compilation des codes sources de 20 bibliothèques avec quatre niveaux d'optimisation, cinq versions de GCC, quatre versions de Clang et pour les plateformes x86 avec 32 et 64 bits. Ce dépôt provient du jeu de données BinKit [85] et les bibliothèques logicielles proviennent des paquets libiconv, coreutils, libtool, gss, gdbm, libtasn1, gsl, libmicrohttpd, osip, readline, gsasl, lightning, recutils, gmp, libunistring et glpk.

Supposons que nous avons les 20 bibliothèques comme ensemble de programmes cibles et que celles-ci sont compilées pour x86 sur 32 bits avec le compilateur GCC en version 6.4 et le niveau d'optimisation -O2. Nous considérons les méthodes suivantes au niveau des fonctions et les adaptons au niveau des programmes comme expliqué précédemment avec la métrique F (voir l'équation 3.1) : Asm2Vec [42], Gemini [154], SAFE [106], αDiff [101]. Nous considérons également LibDB [141], qui est directement conçu pour les bibliothèques, et donc adapté à une quantité importante de code.

TABLE 3.1 – Résultat des recherches de clones.

Méthode	*Precision*@1	Temps total d'exécution (temps de prétraitement inclus)
Asm2Vec [42] †	0,7	35h
Gemini [154] †	1	17h
SAFE [106] †	0,95	160h
αDiff [101] †	1	140h
LibDB [141] †	1	2h
PSS	1	**26s** (dont 26s de pretraitement)

† Temps d'apprentissage non inclus

Résultats Nous rapportons dans la Table 3.1 la précision à la première position (*Precision*@1) moyenne, équivalente à la proportion de recherche de clones réussie, ainsi que le temps total des recherches de clones. Notre contribution PSS est précise et réussit toutes les recherches de clones. La plupart des méthodes utilisant les fonctions peuvent également trouver un clone dans toutes les recherches de clones. Cependant, PSS ne prend que 26s en total, alors que les méthodes par vectorisation de fonctions pure prennent entre 17h, pour Gemini, et 160h, pour SAFE. Même en utilisant une stratégie spécifique aux bibliothèques, LibDB est proche de 2h. De plus, le temps d'exécution de PSS est dû à son prétraitement ; le temps total des calculs de similarité est lui négligeable.

3.6 État de l'art

Programme Compte tenu de ses applications, le domaine de la détection de similarité des programmes a été extrêmement actif au début des années 2000. Les travaux pionniers remontent à Dullien en 2004 [45,46] et étudient des isomorphismes de graphe d'appels ainsi que des couplages entre blocs de bases. Ces deux résultats sont à la base du greffon populaire de comparaison de programmes BinDiff.En 2006, Kruegel et al. [91] ont présenté une approche basée sur la coloration de petits graphes de taille fixe à partir du graphe de flot de contrôle pour identifier les similarités structurelles entre différentes mutations d'un même programme malveillant. En 2008, Gao et al. ont proposé BinHunt [55] pour trouver des différences entre deux versions du même programme. BinHunt utilise l'exécution symbolique avec un solveur de contraintes pour prouver que deux blocs de base implémentent la même fonctionnalité.

D'autres approches incluent, par exemple, les calculs de distances d'édition de graphe [69,89] (GED en anglais) . Avec cette approche, on considère des programmes comme des graphes (de flot de contrôle, d'appels, ou de flux de données) et la distance entre ces graphes est approximée. Dans la même veine, Bruschi et al. [25] abordent le problème de la détection de certains logiciels malveillants à l'intérieur d'un programme en faisant correspondre des graphes de flot de contrôle. Mais cette approche souffre de problèmes sur de longs codes, et n'est donc pas applicable à la recherche sur de grands dépôts de code. On peut considérer que DeepBinDiff de Duan et al. [44] modernise le couplage de BinDiff tout en ajoutant la vectorisation des instructions par l'apprentissage. Cependant, la gestion par blocs de bases sous-jacente en fait un algorithme précis, mais très coûteux en temps. Les techniques de mise en correspondance [7,23,97,152] cherchent à trouver le couplage de coût minimum entre les fonctions des programmes. Plus le coût est faible, plus les programmes sont vus comme proches.

Certains travaux modernes explorent les similarités basées sur des exécutions dynamiques et des observations d'entrée-sortie [5,78,100,108]. Cependant, il est difficile d'explorer complètement l'ensemble des exécutions avec des traces dynamiques - ce qui entraîne une faible précision, et la gestion de larges dépôts de code nécessite d'automatiser la tâche de détection des sources d'entrée et de sortie de tous les programmes dans le dépôt, ce qui peut être difficile. La méthode symbolique de Luo et al. [103] est robuste face aux offuscations simples ainsi qu'aux changements

simples. Cependant, le temps d'exécution symbolique est un problème critique sur les grands programmes et l'offuscation entrave les approches symboliques [11, 114].

Enfin, les méthodes avec N-grammes comparent des séquences d'instructions [68,82,111,130]. Si Exposé [111] considère des trigrammes et des couplages de fonctions, d'autres comme MutantX-S [68] sont plus économes et ne considèrent que des suites d'opcodes. Ces méthodes sont sensibles aux variations introduites par la chaîne de compilation, mais ont l'avantage d'être rapides.

Bibliothèque BAT [64] a proposé trois méthodes pour l'identification des bibliothèques : une qui est basée sur les chaînes de caractères constantes, une sur les algorithmes de compression et une sur les distances d'édition entre des séquences binaires. Les auteurs signalent que les calculs de distance d'édition sont coûteux, tandis que les chaînes peuvent être facilement offusquées. OSSPolice [43] a également développé des mesures de similarité basées sur des chaînes de caractères. Kim et al. [140] ont proposé LibDX qui extrait des chaînes de caractères depuis des sections bien définies du code. Les chaînes de caractères sont mises en correspondance et la statistique TF-IDF est utilisée pour mesurer l'importance des chaînes. Plus récemment, LibDB de Kim et al. [141] combine la vectorisation de fonctions et les couplages avec l'utilisation de chaînes de caractères afin d'élaguer l'ensemble des programmes considérés. La structure spéciale des programmes Java permet d'utiliser des propriétés telles que l'inclusion de classes et de packages [6, 72] afin d'identifier les bibliothèques Android.

Fonction Parallèlement, concernant la recherche de clones de fonctions binaires des méthodes plus coûteuses que la vectorisation existent. Notamment l'analyse dynamique qui cherche à s'appuyer sur la sémantique des codes binaires plutôt que sur leurs propriétés structurelles. Premièrement, BinGo [29] analyse diverses traces d'exécution en mettant en place de l'élagage. Deuxièmement, le travail de Hu et al. [70] émule des fonctions binaires pour créer des signatures sémantiques. Ces travaux souffrent des pièges de l'exécution dynamique : l'exploration est soit imprécise, soit très lente. De plus, appliquer ces méthodes au cas de la similarité de programme est incertain étant donné la taille des codes. Pewny et al. [119] proposent de traduire le code binaire en une représentation intermédiaire. Cette représentation dicte les entrées et les sorties des blocs de base. En utilisant une représentation intermédiaire, plusieurs approches [40,94,120] effectuent une simplification avant la comparaison. Dans la méthode FirmUp [39], le couplage entre des représentations intermédiaires porte sur plusieurs fonctions. Les formules logiques doivent être vectorisées. Plus le segment de code qu'elles représentent est grand, moins précise est la vectorisation. Enfin, des méthodes basées sur des caractéristiques ont été étudiées : Rendez-vous [84] extrait des caractéristiques à différentes granularités, tandis que discovRE [50] et Genius [52] extraient des caractéristiques telles que le nombre d'instructions arithmétiques.

Code source Le problème de la recherche de clones diffère de la similarité des codes sources. En effet, cet autre problème peut être résolu avec différentes structures telles que des arbres syntaxiques abstraits [15,17,156] ou des graphes de dépendance de programme [17]. Il est également

possible de normaliser les instructions et de comparer des fragments de code [17, 77]. La mise en correspondance des jetons, des fragments et des structures est efficace, car il n'y a pas d'étapes d'optimisation du compilateur qui introduirait des variations. De plus, des informations critiques telles que les types sont perdues par compilation, tandis que les dépendances de données sont plus difficiles à récupérer sur les programmes binaires.

Graphe Une question clé dans la similarité des programmes est de trouver comment comparer efficacement les graphes. De nouvelles suggestions pour les similarités des graphes incluent de nouveaux noyaux de graphe [53, 90, 112] et l'utilisation d'apprentissage automatique pour approximer des propriétés difficiles à calculer telles que la distance d'édition de graphe [8, 99, 123]. Récemment, le travail de Bay-Ahmed et al. [2] a introduit une nouvelle métrique de similarité de graphe intégrant des informations spectrales à la fois de la matrice d'adjacence et de la matrice laplacienne. De plus, le travail de Crawford et al. [35] a proposé une analyse spectrale comme métrique de similarité de réseaux sociaux. L'étude de Fyrbiak et al. [54] révèle que l'analyse spectrale peut rivaliser avec des approches plus énergivores telles que la distance d'édition des graphes (GED).

3.7 PSS, une nouvelle analyse spectrale

On peut naturellement voir les programmes comme des graphes, donc une bonne métrique de similarité de graphes est en principe une bonne métrique de similarité de programmes.

La distance d'édition de graphe (GED) est une telle notion [56]. La distance est le coût le plus petit d'un chemin d'édition entre deux graphes, c.a.d. la transformation la plus petite allant d'un des graphes à l'autre. Les opérations d'édition de graphe incluent généralement la suppression ou l'ajout d'un sommet ou d'une arête. Des coûts différents sont associés aux opérations, par exemple le coût de la suppression d'un sommet dépend de son degré. Le principal inconvénient du calcul de GED est que c'est un problème NP-complet [160]. Pire encore, les approximations habituelles ont une complexité de $O(n^3)$ [134] où n est le nombre de sommets du graphe, ce qui est trop coûteux pour les graphes provenant des programmes.

La distance spectrale entre graphes fournit un compromis intéressant, car elle donne une distance entre les graphes à un coût beaucoup plus faible. En effet, grâce à l'algorithme de Lanczos amélioré [113], le calcul du spectre se fait en temps $O(dn^2)$, où d est le degré moyen de G. Une fois que le spectre du programme cible est calculé par le prétraitement, on peut calculer des distances spectrales avec les programmes dans le dépôt en temps $O(n)$.

3.7.1 Théorie spectrale des graphes

L'analyse spectrale des graphes est une méthode utilisée pour étudier les propriétés des graphes en examinant les valeurs propres (ou le *spectre*) des matrices standard associées au graphe, telles que la matrice d'adjacence ou la matrice laplacienne. Des motifs et des structures

au sein du graphe peuvent être identifiés, fournissant des informations clés sur la manière dont les sommets du graphe sont interconnectés. Les distances entre les spectres des graphes sont appelées distances spectrales.

Plus formellement, nous devons calculer la matrice laplacienne [34] du graphe, une matrice semi-définie positive, et obtenir les valeurs propres de cette matrice.

Définition 1. *Un graphe non dirigé $G = (V, E)$ de n sommets est représenté par une matrice d'adjacence A de dimension $n \times n$, où $a_{i,j}$ est 1 si $(V_i, V_j) \in E$ et 0 sinon. Soit d_i, le degré du sommet V_i. La matrice laplacienne est définie par :*

$$L_{i,j} := \begin{cases} d_i & si\ i = j\ et\ d_i \neq 0 \\ -1 & si\ i \neq j\ et\ A_{i,j} \neq 0 \\ 0 & sinon \end{cases}$$

Définition 2. *Une valeur propre λ et un vecteur propre $\vec{u}$ sont des solutions à l'équation $(L - \lambda I)\,\vec{u} = \vec{0}$. Le spectre est l'ensemble Λ des valeurs propres $\{\lambda_1(G), \lambda_2(G), \ldots, \lambda_{|G|}(G)\}$ où $\lambda_1(G) \geq \lambda_2(G) \geq \ldots \geq \lambda_{|G|}(G)$ et où $|G|$ est le nombre de sommets de G.*

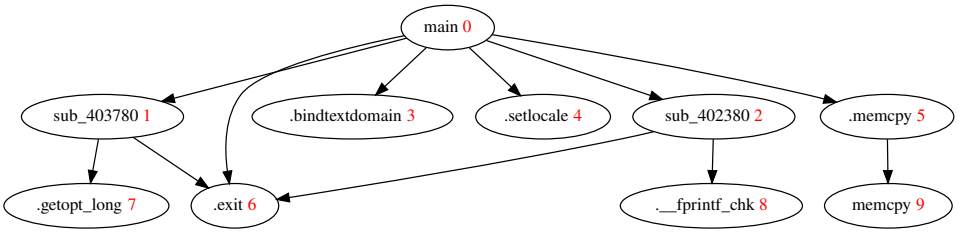


FIGURE 3.2 – Un graphe d'appels de fonctions, considéré non dirigé par l'analyse spectrale.

$$A = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \quad L = \begin{bmatrix} 6 & -1 & -1 & -1 & -1 & -1 & -1 & 0 & 0 & 0 \\ -1 & 3 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 \\ -1 & 0 & 3 & 0 & 0 & 0 & -1 & 0 & -1 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & -1 \\ -1 & -1 & -1 & 0 & 0 & 0 & 3 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \Lambda \approx \begin{bmatrix} 7,10 \\ 4,52 \\ 3,41 \\ 2,54 \\ 1,72 \\ 1 \\ 0,72 \\ 0,58 \\ 0,40 \\ 0 \end{bmatrix}$$

FIGURE 3.3 – Deux graphes G_1 et G_2 différents ayant le même spectre.

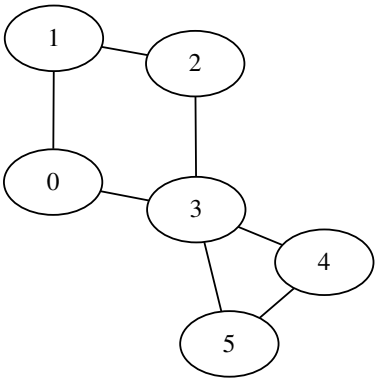


FIGURE 3.4 – G_1.

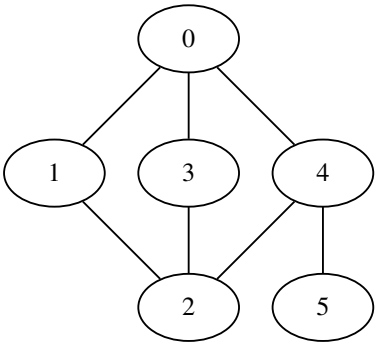


FIGURE 3.5 – G_2.

$$A_1 = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \quad L_1 = \begin{bmatrix} 2 & -1 & -1 & 0 & 0 & 0 \\ -1 & 2 & 0 & -1 & 0 & 0 \\ -1 & 0 & 4 & -1 & -1 & -1 \\ 0 & -1 & -1 & 2 & 0 & 0 \\ 0 & 0 & -1 & 0 & 2 & -1 \\ 0 & 0 & -1 & 0 & -1 & 2 \end{bmatrix} \quad \Lambda_1 \approx \begin{bmatrix} 5,24 \\ 3 \\ 3 \\ 2 \\ 0,76 \\ 0 \end{bmatrix}$$

$$A_2 = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \quad L_2 = \begin{bmatrix} 3 & -1 & -1 & -1 & 0 & 0 \\ -1 & 2 & 0 & 0 & -1 & 0 \\ -1 & 0 & 2 & 0 & -1 & 0 \\ -1 & 0 & 0 & 3 & -1 & -1 \\ 0 & -1 & -1 & -1 & 3 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 \end{bmatrix} \quad \Lambda_2 \approx \begin{bmatrix} 5,24 \\ 3 \\ 3 \\ 2 \\ 0,76 \\ 0 \end{bmatrix}$$

Théorème 1. *Soit $e(G)$ le nombre d'arêtes de G, alors $\sum_{i=1}^{|G|} \lambda_i(G) = 2e(G)$.*

Les valeurs propres de L sont positives et 0 est une valeur propre de L. En outre, le théorème 1 peut être facilement prouvé en utilisant la trace des matrices [95]. L'intuition derrière l'usage du spectre pour comparer des programmes est que deux graphes isomorphes ont le même spectre. Cependant, il existe des graphes distincts qui partagent le même spectre, ils sont dits cospectraux (voir un exemple en Figure 3.3).

Définition 3. *La distance spectrale [80] entre deux graphes G_1, G_2 de même nombre de sommets $n = |G_1| = |G_2| =$ est $\sum_1^n |\lambda_i(G_1) - \lambda_i(G_2)|$.*

La distance spectrale généralisée est définie par : $\sum_{i=1}^{min(|G1|,|G2|)} |\lambda_i(G_1) - \lambda_i(G_2)|$.

Néanmoins, le spectre peut être utilisé pour comparer les similitudes des graphes. Les travaux de Wilson et Zhu [150] montrent qu'une distance spectrale euclidienne est empiriquement reliée à

la distance d'édition entre les graphes. De plus, de nombreux théorèmes [59] décrivent comment les valeurs propres restent proches suite à des modifications des graphes.

Théorème 2. *Soit G' une copie d'un graphe G auquel on a retiré un sommet de degré r. Pour tout i tel que $1 \leq i \leq n-1$, $\lambda_i(G) \geq \lambda_i(G') \geq \lambda_{i+r}(G)$.*

Corollaire 2.1. *La distance spectrale généralisée entre G et G' est alors égale à $2r$.*

Démonstration. D'après le théorème 2, $\lambda_i(G) \geq \lambda_i(G')$ pour tout i tel que $1 \leq i \leq |G| - 1$.

On peut donc déduire que :

$$\sum_{1}^{min(|G|,|G'|)} |\lambda_i(G) - \lambda_i(G')| = \sum_{1}^{|G|-1} |\lambda_i(G) - \lambda_i(G')| = \sum_{1}^{|G|-1} \lambda_i(G) \quad - \quad \sum_{1}^{|G|-1} \lambda_i(G')$$

Or puisque 0 est une valeur propre de toute matrice laplacienne, $\lambda_{|G|}(G) = 0$.

On peut alors utiliser le théorème 1 pour prouver que :

$$\sum_{1}^{|G|-1} \lambda_i(G) \quad - \quad \sum_{1}^{|G|-1} \lambda_i(G') = \sum_{1}^{|G|} \lambda_i(G) \quad - \quad \sum_{1}^{|G'|} \lambda_i(G') = 2e(G) - 2e(G')$$

Or, $e(G') = e(G) - r$ puisqu'on a retiré un sommet de degré r.

Donc la distance spectrale généralisée entre G et G' est de $2e(G) - 2(e(G) - r) = 2r$ □

Le théorème 2 prouve que la suppression d'un sommet a un impact sur le spectre dépendant du degré de ce sommet. De plus, le corollaire 2.1 implique qu'après la suppression d'un sommet, la distance spectrale généralisée avec le graphe d'origine est de deux fois le degré du sommet, ce qui est très semblable à un coût d'édition de graphe.

3.7.2 PSS, l'analyse spectrale des programmes

L'analyse spectrale est une bonne idée pour comparer les graphes, car elle fournit des mesures quantitatives, telles que les distances spectrales, qui peuvent être utilisées pour comparer des propriétés clés des graphes comme la connectivité, la structure et la distribution. Cette approche permet également de passer outre la taille des graphes afin de réaliser des comparaisons équitables entre des graphes de tailles variables.

Cependant, le calcul du spectre d'un graphe est cubique par rapport au nombre de sommets. Par conséquent, appliquer l'analyse spectrale à l'ensemble du graphe de flot de contrôle d'un programme est trop coûteux. De plus, le graphe de flot de contrôle lui-même n'est pas stable par rapport au compilateur, à l'architecture, aux optimisations ou encore aux offuscations.

En conséquence, notre idée clé est qu'un programme a plus de structure qu'un simple graphe : il existe un graphe d'appels des fonctions tandis que les fonctions locales ont leur propre graphe de flot de contrôle. Nous tirons parti de cette structure hiérarchique pour élaborer une métrique de similarité rapide et stable appelée *similarité spectrale de programme* (PSS).

La méthode PSS se base sur la combinaison de deux critères. Le premier est la distance spectrale entre les graphes d'appel, incluant les appels internes et externes. Les graphes d'appel sont utiles pour un certain nombre de tâches. Par exemple, GraphEvo [147] a été capable de comprendre l'évolution du logiciel à travers les graphes d'appel. De plus, les optimisations du compilateur ont relativement peu d'effet sur le graphe d'appels. Le second est une analyse spectrale grossière des graphes de contrôle des fonctions, en considérant simplement leur nombre d'arêtes, car il est lié aux valeurs spectrales (voir le théorème 1 plus haut).

La méthode PSS est divisée en deux étapes : l'étape de prétraitement de la cible, qui est faite une fois pour toutes, et l'étape de vérification de la similarité, qui est effectuée pour chaque calcul de similarité.

Prétraitement Soit un programme P, le prétraitement commence par la construction du graphe d'appels de fonctions CG de P, y compris les appels locaux et externes (API). Un exemple de graphe d'appels de fonctions est donné dans la Figure 3.2. Il contient des appels externes tels qu'un appel à `mempcpy` ainsi que des fonctions locales telles que `sub_403780`.

À partir de cela, nous extrayons deux vecteurs clés $(\vec{v}, \vec{w})$ de P comme suit :

- Depuis une version non dirigée du graphe d'appels CG, nous calculons le spectre $\Lambda = \{\lambda_1(CG), \ldots, \lambda_n(CG)\}$, et le spectre normalisé du graphe d'appels $\vec{v} := \frac{\Lambda}{\|\Lambda\|_2}$;
- Nous calculons le nombre d'arêtes $E = (e_1, e_2, \ldots, e_k)$ de chaque graphe de flux de contrôle F_i des fonctions locales par ordre décroissant, et nous normalisons E comme précédemment : $\vec{w} := \frac{E}{\|E\|_2}$.

Pour rappel, $||\cdot||_2$ est la norme euclidienne. Nous normalisons les vecteurs $\vec{v}$ et $\vec{w}$ pour faire face aux différences de taille des programmes.

Calcul de similarité Pour deux programmes, P_0 et P_1, l'étape de prétraitement a calculé les vecteurs $(\vec{v_0}, \vec{w_0})$ à partir de P_0, et $(\vec{v_1}, \vec{w_1})$ à partir de P_1. Le calcul de similarité produit un indice de similarité par la moyenne de deux métrique. La première métrique (3.2) est liée aux graphes d'appel, tandis que la seconde (3.3) est liée aux graphes de flot de contrôle des fonctions. Enfin, la métrique de similarité PSS est définie comme la moyenne des deux mesures ci-dessus (équation 3.4).

$$simCG(P_0, P_1) := \sqrt{2} - \sqrt{\sum_{i=0}^{min(|\vec{v_0}|,|\vec{v_1}|)} (v_{0,i} - v_{1,i})^2} \tag{3.2}$$

$$simCFG(P_0, P_1) := \sqrt{2} - \sqrt{\sum_{i=0}^{min(|\vec{w_0}|,|\vec{w_1}|)} (w_{0,i} - w_{1,i})^2} \tag{3.3}$$

$$PSS(P_0, P_1) := \frac{simCG(P_0, P_1) + simCFG(P_0, P_1)}{2\sqrt{2}} \tag{3.4}$$

PSSO, la version optimisée Nous avons découvert que le prétraitement de PSS pouvait être relativement long sur des programmes contenant beaucoup de fonctions. La Section 4.5.3 explique que, sur notre jeu de données Windows, calculer toutes les valeurs propres d'un graphe d'appels prend $16,95$ secondes par recherche de clones. Afin de régler ce problème, plutôt que de calculer le spectre complet Λ, nous proposons de ne calculer que les K plus grandes valeurs propres, de telle sorte que $\Lambda = \{\lambda_1(CG), \ldots, \lambda_K(CG)\}$.

La Figure 3.6 présente les temps de prétraitement ainsi que les scores de précision (défini dans la Section 4.3) pour différentes valeurs de K depuis 30 jusqu'à 180 sur notre jeu de données Windows (décrit dans la Section 4.5.1). Nous remarquons que les temps moyens de prétraitement grandissent vite avec K, passant de $0,06$ seconde à $1,31$ seconde. D'un autre côté, il y a peu de changement sur le score de précision entre 50 et 150 puisqu'il varie entre $0,4657$ et $0,4664$. Nous sélectionnons 100 comme valeur de K puisque le temps de prétraitement n'est que de $0,39$ seconde en moyenne alors que le score de précision est déjà $0,4661$.

Nous proposons donc PSSO, une version optimisée de PSS qui calcule seulement les $K = 100$ plus grandes valeurs propres.

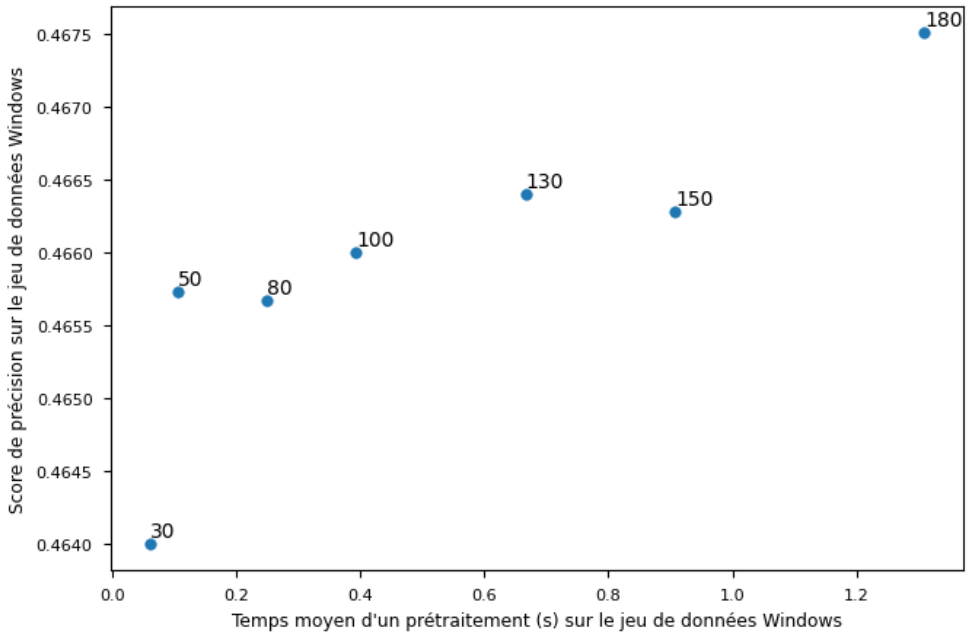


FIGURE 3.6 – Impact sur le jeu de données Windows du nombre K de plus grandes valeurs propres calculées par la version optimisée de PSS.

Complexité en temps Rappelons qu'une base de données de programmes prétraités est appelée un dépôt. Un programme cible inconnu est d'abord prétraité, puis des calculs de similarité avec chaque programme du dépôt sont effectués à partir des caractéristiques prétraitées. Il est clair que le temps d'exécution de la requête dépend multiplicativement de la taille du dépôt. En d'autres termes, si le temps d'exécution d'un calcul de similarité est $T(n)$, le temps d'exécution

du prétraitement est $P_T(n)$ et la taille du dépôt est M, alors le temps d'une requête est borné par $M \times T(n) + P_T(n)$. Par conséquent, toutes les méthodes ayant un temps de calcul de similarité plus grand que linéaire sont trop lentes, et cette observation sera confirmée par nos expériences.

Les graphes sont creux dans notre domaine d'application, ce qui permet un calcul rapide du spectre. Néanmoins, la complexité du prétraitement de la requête, décrite dans la Section 3.7.2, est toujours de $O(dn^2)$, où n est le nombre de fonctions et d est le nombre moyen d'appels par fonction. Cependant, le temps de calcul d'une similarité, décrit dans la Section 3.7.2, est de $O(n)$. De plus, le temps d'exécution du prétraitement de la version optimisée de PSS (PSSO) est réduit à $O(dn)$.

Comparaisons avec les travaux antérieurs Nous présentons dans la Table 3.2 les complexités en temps des autres méthodes. La complexité en temps de PSS est en contraste avec les méthodes par vectorisation de fonctions dont le temps de calcul d'un indice de similarité est de $O(n^2)$, en utilisant une adaptation directe (voir la Section 3.5 pour plus de détails). De plus, DeepBinDiff effectue un couplage minimal entre les blocs de base, ce qui induit une complexité de $O(n^3m^3)$. L'approximation de la distance d'édition de graphe de SMIT [69] et le couplage de Xu et al. [152] sont encore plus coûteux, avec une complexité en temps de $O(n^4)$. En revanche, MutantX-S [68], qui est pensé pour supporter de large dépôt, a une complexité de $O(1)$. Cependant, nos expériences montrent que la robustesse de MutantX-S n'est pas satisfaisante.

Table 3.2 – Complexité en temps des recherches de clones de programmes.

Méthode	Classe	Calcul de Similarité†	Prétraitement‡
SMIT [69]	GED	$O(n^4)$	$O(dn)$
CGC [152]	Couplage	$O(n^4)$	$O(dn)$
MutantX-S [68]	**N-gramme**	$O(1)$	$O(i)$
Asm2Vec [42]	Fonction	$O(n^2)$	$O(n)$
Gemini [154]	Fonction	$O(n^2)$	$O(n)$
SAFE [106]	Fonction	$O(n^2)$	$O(n)$
αDiff [101]	Fonction	$O(n^2)$	$O(n)$
LibDX [140]	Chaînes	$O(s)$	$O(s)$
LibDB [141]	Chaînes et fonction	$O(n^2 + s)$	$O(s + n)$
DeepBinDiff [44]	Apprentissage et couplage	$O(n^3m^3)$	Absent
PSS	**Spectral**	$O(n)$	$O(dn^2)$
PSSO	**Spectral**	$O(n)$	$O(dn)$

n : # de fonctions, i : # d'instructions , s : # de chaînes de caractères constantes
d : # moyen d'appels par fonction, m : # de blocs de base par fonction
† Entre deux programmes
‡ Une fois pour toute la recherche de clones

Chapitre 4

La recherche de clones : étude systématique

Le Chapitre 4 est une évaluation rigoureuse de PSS et de 15 outils de l'état de l'art permettant de rechercher des clones. L'objectif est de trouver les méthodes les plus rapides, précises et robustes. La pertinence des identificateurs littéraux (tels que des chaînes littérales) et leurs limites obligent à analyser les méthodes les utilisant séparément. Dans le premier cas, nous mettons au point deux nouvelles heuristiques. Une évaluation préliminaire sur un petit jeu de données est riche d'enseignement sur les avantages et les inconvénients des classes de méthodes par rapport à deux scénarios. Ensuite, nous éliminons les méthodes trop lentes pour être utilisées à grande échelle afin de mettre à l'échelle la recherche de clones sur 97 760 programmes Linux, 19 959 programmes malveillants ciblant les objets connectés et 84 992 programmes Windows. PSS, et en particulier sa version optimisée PSSO, est démontré comme un bon compromis entre la vitesse et la précision tout en étant très robuste, même en cas de changement d'architecture et d'offuscation. De plus, si les identificateurs littéraux sont disponibles, une heuristique très simple comparant les identificateurs est la méthode la plus précise bien qu'elle soit lente.

4.1 Objectif de recherche

Nous souhaitons évaluer les différentes méthodes de recherche de clones selon trois dimensions. Premièrement, la rapidité, c.a.d. la vitesse d'exécution d'une recherche de clones. Deuxièmement, la précision, c.a.d. la qualité des résultats produits. Enfin, la robustesse, c.a.d. la stabilité de la précision face à des changements de chaîne de compilation ou de version de code source.

Nous considérons donc les questions de recherche suivantes :

RQ1 Quelles sont les méthodes les plus rapides pour la recherche de clones ?

RQ2 Quelles sont les méthodes les plus précises pour la recherche de clones ?

RQ3 Quelles sont les méthodes les plus robustes pour la recherche de clones ?

RQ4 Quel est l'impact des composantes de PSS ?

4.2 Autres approches

Nous considérons 15 outils de l'état de l'art, trois points de comparaison et deux nouvelles heuristiques basées sur des identificateurs littéraux (telles que des chaînes littérales ou les noms de fonctions externes). Un total de 8 des outils a été adapté **(A)** à la recherche de clones de programmes, car leur objectif principal était différent (p. ex., la recherche de clones de fonctions). De plus, 11 des outils ont dû être réimplantés **(R)**, car l'implémentation originale était indisponible ou parce qu'il était difficile d'utiliser l'implémentation originale dans le cadre de la recherche de clones de programmes. Comme souligné par Marceli et al. [104] les codes sources sont rarement disponibles, et même lorsqu'ils le sont, ils sont souvent incomplets.

Nous présentons dans la Table 4.1 la liste des outils et les caractéristiques des différentes méthodes considérées ici. Nous indiquons la complexité en temps d'un calcul de l'indice de similarité entre deux programmes. Nous notons le nombre de fonctions d'un programme par n, le nombre de blocs de base dans un graphe de flot de contrôle d'une fonction par m, et le nombre d'identificateurs littéraux dans un programme par s. Enfin, nous indiquons si la méthode nécessite des identificateurs littéraux, une phase d'apprentissage ou une classification manuelle des mnémoniques.

Points de comparaison Nous utilisons comme points de comparaison des heuristiques simplistes telles que B_{size}, la taille du programme, et D_{size}, la taille du programme désassemblé. Par exemple, la métrique de similarité B_{size} est définie comme $B_{size}(a, b) := -|a - b|$, où a et b sont les tailles des programmes en octets. Nous considérons également une métrique de similarité grossière du graphe d'appels. Soit n_1 et e_1 (resp. n_2 et e_2) les nombres de sommets et d'arêtes du premier (resp., second) graphe d'appels. Alors, la métrique de similarité Shape est définie comme : $Shape(n_1, e_1, n_2, e_2) := \frac{\min(n_1,n_2)}{\max(n_1,n_2)} \times \frac{\min(m_1,m_2)}{\max(m_1,m_2)}$.

Méthodes spectrales standard À partir de la méthode spectrale développée par Fyrbiak et al. [54], nous dérivons deux méthodes :

- ASCG **(A)** **(R)** : La première ASCG est basée sur le graphe d'appels. Soit X et Y, les deux spectres des matrices laplaciennes des deux graphes d'appel. Après une normalisation $X' := X/X_0$, $Y' := Y/Y_0$, nous calculons : $ASCG(X', Y') := -\sum_{i=0}^{min(|X'|,|Y'|)} |X'_i - Y'_i|$;
- ASCFG **(A)** **(R)** : La seconde ASCFG est semblable à ASCG, mais basée sur le graphe de flot de contrôle. Au lieu du spectre du graphe d'appel, nous sélectionnons comme vecteurs X et Y les 1000 premières valeurs propres du graphe de flot de contrôle (CFG) complet du programme. Les graphes de flot de contrôle (CFG) sont extraits avec le désassembleur Gorille [12] muni d'une analyse symbolique et concrète afin de parcourir toutes les exécutions possibles du programme [22]. Les graphes sont réduits par Gorille afin de supprimer les instructions séquentielles pour ne garder que le flot de l'exécution.

12. `https://cyber-detect.com/gorille/`

TABLE 4.1 – Methodes comprises dans notre étude systématique.

Méthode	Classe	A	R	Calcul de similarité	IL	Appr.	Class.
B_{size}	**Point de comparaison**			$O(1)$			
D_{size}	**Point de comparaison**			$O(1)$			
Shape	**Point de comparaison**			$O(1)$			
ASCG [54]	**Spectrale**	×	×	$O(n)$			
ASCFG [54]	**Spectrale**	×	×	$O(1)$			
GED-0 [129]	GED	×	×	$O(n^3)$			
Asm2Vec [42]	Fonction	×		$O(n^2)$		×	
Gemini [154]	Fonction	×		$O(n^2)$		×	×
SAFE [106]	Fonction	×		$O(n^2)$		×	
MutantX-S [68]	**N-gramme**		×	$O(1)$			
DeepBinDiff [44]	Couplage			$O(n^3m^3)$		×	
PSS	**Spectrale**			$O(n)$			
PSS$_O$	**Spectrale**			$O(n)$			
ISO [7]	Couplage		×	$O(n^2)$	×		
CGC [152]	Couplage		×	$O(n^4)$	×		
GED-L [54]	GED	×	×	$O(n^3)$	×		
SMIT [69]	GED		×	$O(n^4)$	×		
αDiff [101]	Fonction	×	×	$O(n^2)$	×	×	
LibDX [140]	Chaînes		×	$O(s)$	×		
LibDB [141]	Chaînes et fonction		×	$O(n^2 + s)$	×	×	×
StringSet	Chaînes			$O(s)$	×		
FunctionSet	**Chaînes**			$O(n)$	×		

A : Adapté pour la recherche de clones de programmes, R : Réimplémenté

IL : Des identificateurs littéraux sont utilisés

Appr. : Phase d'apprentissage, Class. : Classification manuelle des mnémoniques

Distances d'édition de graphes Soit $G1$ et $G2$ les deux graphes d'appels de fonctions issus des programmes dont nous calculons un indice de similarité. Nous implémentions plusieurs méthodes basées sur des distances d'éditions entre $G1$ et $G2$:

- GED-0 **(A) (R)** : Tout d'abord, nous implémentons GED-0, une approximation standard de la distance d'édition. L'algorithme est semblable au travail pionnier de Sanfeliu et Fu [129]. Le principe est de calculer le couplage minimal entre les sommets des deux graphes en considérant uniquement les nombres de successeurs et prédécesseurs des sommets. Soit $s_G(a)$ et $p_G(a)$, les nombres de successeurs et prédécesseurs du sommet a dans le graphe G. Le coût d'un couplage entre un sommet u de $G1$ et un sommet v de $G2$ est $|s_{G1}(u) - s_{G2}(v)| + |p_{G1}(u) - p_{G2}(v)|$. Le coût de la suppression d'un sommet u de $G1$ est $s_{G1}(u) + p_{G1}(u) + 1$ (de même pour un sommet v de $G2$).

  L'algorithme hongrois [92] permet de calculer le couplage minimal des sommets de $G1$ avec les sommets de $G2$ afin d'approximer grossièrement la distance d'édition entre $G1$ et $G2$. Nous utilisons l'implémentation de l'algorithme hongrois proposée par le paquet python `lapjvde` ;
- GED-Labels **(A) (R)** : Deuxièmement, nous implémentons GED-Labels, une approximation de distance d'édition où ce sont les étiquettes des nœuds qui indiquent le coût de couplage des sommets. L'algorithme est présenté par Fyrbiak et al. [54]. Il s'applique dans leur cas à des circuits intégrés. Dans notre application, les étiquettes sont des ensembles de noms de fonctions externes. Soit $eti(a)$ l'ensemble des appels externes d'un sommet a dans G. Le coût d'un couplage entre un sommet u de $G1$ et un sommet v de $G2$ est $|eti_{G1}(u)| + |eti_{G2}(v)| - 2 \times |eti_{G1}(u) \cap eti_{G2}(v)|$. Le coût de la suppression d'un sommet u de $G1$ est $|eti_{G1}(u)|$ (de même pour un sommet v de $G2$).

  Comme précédemment, l'implémentation de l'algorithme hongrois du paquet python `lapjvde` permet d'approximer une distance d'édition entre $G1$ et $G2$;
- SMIT **(R)** : Troisièmement, nous implémentons un calcul de distance d'édition spécifique pour les programmes de Hu et al. [69] appelé SMIT [13]. Les fonctions sont tout d'abord couplées si la distance d'édition entre leurs mnémoniques est basse. Bien sûr pour être couplées deux fonctions doivent être présentes à la fois dans $G1$ et dans $G2$, elles sont alors fusionnées pour ne former qu'une. Puis, les fonctions qui font les mêmes appels sont également couplées. Les fonctions couplées sont alors retirées. Soit $o_G(a)$ et $i_G(a)$, les ensembles de successeurs et de prédécesseurs du sommet a dans le graphe G. Le coût de la suppression d'une fonction u de $G1$ est $|s_{G1}(u)| + |i_{G1}(u)| + 1$ (de même pour une fonction v de $G2$). Le coût d'un couplage entre une fonction u de $G1$ et une fonction v de $G2$ est : $|o_{G1}(u)| + |o_{G2}(v)| + |i_{G1}(u)| + |i_{G2}(v)| - 2 \times (|o_{G1}(u) \cap o_{G2}(v)| + |i_{G1}(u) \cap i_{G2}(v)|)$.

  Une variante de l'algorithme hongrois pour le couplage a été inventée pour SMIT afin

13. Nous n'intégrons pas l'arbre d'indexation de SMIT utilisant des heuristiques. Premièrement, celui-ci était conçu pour des scénarios où de nombreux clones étaient présents dans le dépôt. Deuxièmement, bien que l'utilisation d'un arbre d'indexation améliore la vitesse d'exécution, cela reste moins précis que le calcul de tous les indices de similarités.

de fournir une autre approximation de la distance d'édition entre $G1$ et $G2$. Lorsque les fonctions u et v sont couplées, les paires de fonctions n, m telles que n (resp. m) est appelée par u (resp. v) bénéficient d'une réduction du coût de couplage proportionnelle à la distance d'édition entre leurs mnémoniques.

Couplages Soit $G1 = (V1, E1)$ et $G2 = (V2, E2)$ les deux graphes d'appels de fonctions issus des programmes dont nous calculons un indice de similarité. Bien que les approximations de la distance d'édition reposent de fait déjà sur le couplage de fonctions, elles assignent des coûts à ces couplages et cherchent un coût total minimal. D'autres méthodes par couplages existent où plus le couplage est important, plus les programmes sont considérés comme similaires :

- ISO **(R)** : Nous reproduisons le test d'isomorphisme ISO du travail de Bai et al. [7]. Cet algorithme consiste simplement à trouver pour chaque fonction d'un programme, celle avec le même nom dans l'autre programme. Néanmoins, dans le cas des programmes dépouillés de symboles, ce processus ne fonctionne pas. De plus, cet algorithme n'est adapté que dans le cas d'un isomorphisme parfait entre les programmes. Lorsqu'une fonction ne peut être couplée, le comportement de l'algorithme est indéfini. Nous avons décidé d'arrêter l'algorithme lorsque cela arrive. Comme nous le verrons plus tard, ce choix conduit l'algorithme à toujours répondre que deux programmes sont totalement différents. Cependant, continuer aurait fait que l'algorithme réponde toujours que deux programmes sont parfaitement similaires ;
- CGC **(R)** : Nous reproduisons également l'algorithme de couplage CGC de Xu et al. [152]. Cet algorithme demande tout d'abord une classification manuelle des mnémoniques en 15 catégories. En tout, nous avons classifié 382 mnémoniques différentes.

  Comme précédemment, les fonctions couplées doivent être présentes à la fois dans $G1$ et dans $G2$, et elles sont fusionnées pour ne former qu'une. La première phase de l'algorithme consiste à réaliser des couplages initiaux. Les fonctions externes présentes dans les deux programmes sont couplées. Puis, les fonctions locales qui appellent plus de deux fonctions externes communes sont couplées. Enfin, les fonctions sont couplées si elles franchissent des seuils de similarités par rapport aux mnémoniques, aux nombres d'instructions, et aux nombres d'appels de fonctions. Après un premier travail sur les programmes venant du paquet Coreutils, nous réglons ces paramètres à $0,45$, $0,18$ et $0,3$ respectivement. La seconde phase de l'algorithme consiste à propager les couplages. Soit u et v deux fonctions déjà couplées, ainsi que n (resp. m) une fonction appelée par u (resp. v). Les fonctions n et m sont couplées si les mnémoniques sont similaires avec un seuil de $0,50$. La dernière phase de l'algorithme consiste à ne garder que les fonctions couplées et à compter le nombre d'appels restants entre fonctions c. La métrique de similarité est alors simplement : $\frac{2 \times c}{|E1|+|E2|}$.

MutantX-S (R) Nous reproduisons MutantX-S de Hu et al. [68]. Son prétraitement produit comme caractéristiques un vecteur de dimension 4096 à partir du programme cible. Soit u et v les deux vecteurs obtenus à partir de deux programmes. La métrique de similarité de MutantX-S est simplement : $||u - v||_2$.

La première phase du prétraitement obtient la séquence des *opcodes* présents dans les instructions du programme. Les opcodes sont la partie du code binaire de l'instruction qui précise quelle est l'instruction à exécuter. Par exemple, l'opcode `88` (noté en hexadécimal) indique que l'instruction `886521` déplace des données de registres dont la taille est d'un octet. La mnémonique de l'instruction est simplement `mov`. Suivant l'architecture matérielle, les opcodes peuvent être placés à différents endroits ainsi qu'occuper une place plus ou moins importante dans le code binaire de l'instruction. Pour les extraire, il faut consulter le manuel spécifique à l'architecture matérielle. Par exemple, pour l'architecture x86 il faut prendre en compte que différents drapeaux peuvent être placés avant chaque opcode et que les opcodes ont des tailles variables [74]. Alors que l'article original n'avait prévu que l'utilisation pour l'architecture matérielle x86, nous implémentons également l'extraction des opcodes pour les architectures ARM, MIPS, SuperH (SH-3 et SH-4), PowerPC, SPARC, ARC (ARCompact et ARC-V2), Motorola 68040 et IBM System/390.

La seconde phase génère les occurrences des suites d'opcodes dans le programme. Pour cela, il suffit d'énumérer tous les 4-grammes présents dans la séquence des opcodes. Les 4-grammes sont toutes les sous-séquences de 4 éléments d'une séquence.

La troisième phase hache les occurrences des 4-grammes pour produire le vecteur représentant le programme. Soit m ce vecteur de dimension 4096. Le hachage utilise une fonction de hachage h modulo 4096. Pour chaque 4-gramme g d'occurrence o, la position $h(g)$ du vecteur m est incrémentée de o.

Vectorisation de fonctions La vectorisation des fonctions est du ressort de l'apprentissage automatique et inspirée du traitement du langage automatique des langues. L'un des avantages principaux de cette méthode est son ambition d'encoder des relations sémantiques entre les fonctions. En effet, deux fonctions qui accomplissent des tâches similaires devraient avoir une représentation vectorielle similaire. Pour y arriver, on peut utiliser des techniques d'apprentissage en profondeur pour apprendre une représentation vectorielle des fonctions. Les réseaux de neurones convolutifs (CNN) et les réseaux de neurones récurrents (RNN) sont parmi les modèles plus couramment utilisés à cet effet. Ces modèles sont entraînés sur d'importants jeux de données et apprennent à encoder les fonctions en vecteurs de manière à optimiser une certaine tâche d'apprentissage, le plus souvent simplement prédire si deux fonctions ont été compilées avec le même code source. Une fois un modèle entraîné, on peut l'utiliser pour transformer une nouvelle fonction en vecteur. En calculant la distance entre ce vecteur avec ceux des fonctions connues , nous pouvons identifier rapidement la fonction.

Adaptation des méthodes de vectorisation de fonctions La Section 3.4.2 décrit comment nous adaptons les méthodes de vectorisation de fonctions au cadre de la recherche de clones de programmes. En utilisant un outil de vectorisation donné, nous transformons un programme en un ensemble de vecteurs, chacun représentant une fonction du programme. Nous utilisons une métrique de similarité simple pour comparer deux ensembles de vecteurs a et b : $-\sum_{x\in a} \min_{y\in b} \|x-y\|_2$. Nous considérons quatre outils de vectorisations différents :

- Asm2Vec **(A)** : Le premier est Asm2Vec [42]. Nous employons une stratégie d'apprentissage inspirée de l'article original. Chaque champ de test dans le second scénario donne lieu à un apprentissage. En effet, les programmes ayant un niveau d'optimisation (ou de version) le plus élevé font office d'ensemble d'entraînement. Les autres programmes ne sont utilisés que pour obtenir des vecteurs. Chaque apprentissage dure 50 époques avec un taux d'apprentissage initial de $0,05$, une fenêtre de taille 2, 25 échantillons négatifs pour chaque échantillon positif et 10 marches aléatoires. La dimension finale des vecteurs est de 200 ;
- Gemini **(A)** : Nous ajoutons ensuite Gemini de Xu et al. [154], de façon à le surévaluer. Nous construisons une version de notre jeu de données Basique conservant les noms des fonctions et les utilisons comme étiquette. Nous apprenons sur ce jeu d'entraînement pendant 50 époques avec cinq itérations, un taux d'apprentissage de $0,0001$ et une profondeur du réseau de deux. Puis, nous calculons les vecteurs d'après le modèle obtenu sur le jeu de donnée sans les noms des fonctions. La dimension des vecteurs est de 64 ;
- SAFE **(A)** : De plus, nous adaptons la méthode de Massarelli et al. [106], SAFE. Nous utilisons un modèle préentraîné mis à disposition par l'un des auteurs [14]. La dimension des vecteurs est de 100 ;
- αDiff **(A) (R)** : Enfin, nous réimplémentons αDiff de Liu et al. [101]. Cette méthode est spécialisée dans la recherche de clones de fonctions dont ce sont les versions qui diffèrent. Nous échantillonnons 25% du jeu de données d'αDiff [15] comme notre jeu d'entraînement. Nous apprenons sur ce jeu d'entraînement pendant 20 époques, avec un taux d'apprentissage de $0,001$ et nous réglons le facteur d'oubli à $0,9$. À chaque époque, 100 paires positives (des fonctions clones) sont utilisées ainsi que 200 paires négatives. αDiff propose une distance entre les fonctions qui incorpore les noms des fonctions externes et les degrés des fonctions dans le graphe d'appels.

DeepBinDiff DeepBinDiff de Duan et al. [44] essaye de faire correspondre les blocs de base de deux programmes en s'appuyant à la fois sur l'apprentissage pour vectoriser les instructions et sur les graphes de flot de contrôle. Plus le couplage entre les blocs de bases de deux programmes est abouti, plus les programmes sont similaires. Bien que nous ayons pu constater que cet outil est précis, il est uniquement considéré dans l'étude préliminaire en raison de son temps d'exécution très important.

14. `https://github.com/facebookresearch/SAFEtorch`
15. `https://twelveand0.github.io/AlphaDiff-ASE2018-Appendix`

LibDX (R) Nous reproduisons LibDX de Kim et al. [140]. La méthode extrait des chaînes littérales depuis les sections en écriture seule des programmes. Ensuite les chaînes littérales sont standardisées en retirant des spécificités superflues telles que la différence entre les minuscules et les majuscules. Ces chaînes littérales sont comparées via la recherche de correspondances et en utilisant la statistique TF-IDF [139] pour mesurer l'importance de ces correspondances.

LibDB (R) Nous reproduisons LibDB de Kim et al. [141]. Cette méthode combine la vectorisation des fonctions de Gemini avec l'identification de chaînes littérales communes comme premier filtre. Tout d'abord, LibDB établit deux listes de candidats : une grâce aux correspondances des chaînes littérales et une avec une recherche des vecteurs les plus proches de ceux des fonctions du programme recherché. Ensuite, LibDB effectue des calculs des indices de similarité entre le programme recherché et les candidats. Ces calculs sont relativement complexes. Premièrement, un couplage des fonctions communes entre les deux programmes est effectué. Deuxièmement, les graphes d'appels de fonctions sont réduits aux fonctions couplées, tout en maintenant la connectivité des graphes. Enfin, la proportion d'arêtes communes dans les deux graphes d'appels donne l'indice de similarité des programmes. Nous avons réimplanté LibDB avec notre modèle de Gemini déjà entraîné et ScaNN [58] comme outil de recherche des vecteurs les plus proches.

Noubelles heuristiques Les outils précédents qui utilisent les identificateurs littéraux sont complexes et utilisent généralement d'autres ingrédients que ceux-ci. Pour permettre de juger si cette complexité est nécessaire, nous introduisons deux heuristiques basées entièrement sur les identificateurs littéraux :

- FunctionSet : Xu et al. [152] décrivent une méthode simple qui commence par coupler des fonctions entre deux programmes en utilisant uniquement les noms des fonctions externes et la similarité des mnémoniques. Nous simplifions cette idée et inventons la métrique de similarité FunctionSet, qui calcule l'indice de similarité de Jaccard [16] entre les noms des fonctions externes. Soit E_a, l'ensemble des noms des fonctions externes d'un programme a. La métrique de similarité entre deux programmes a et b est :

$$FunctionSet(a, b) := \frac{|E_a \cap E_b|}{|E_a \cup E_b|}$$

- StringSet : De même, nous inventons une métrique simple qui incorpore toutes les chaînes littérales à l'intérieur des programmes. Pour cela, nous utilisons le programme strings qui provient du paquet Binutils. Pour détecter les chaînes littérales, il inspecte le code binaire à la recherche de suites de caractères supérieurs. Soit C_a, l'ensemble de toutes les chaînes littérales d'un programme a. La métrique de similarité entre deux programmes a et b est :

$$StringSet(a, b) := \frac{|C_a \cap C_b|}{|C_a \cup C_b|}$$

16. `https://fr.wikipedia.org/wiki/Indice_et_distance_de_Jaccard`

4.3 Méthodologie

Notre étude porte sur les 20 méthodes de recherche de clones de programmes introduites dans la Section précédente ainsi que sur nos deux propositions PSS et PSSO. Nous réalisons une évaluation systématique des méthodes en faisant une distinction entre les méthodes nécessitant des identificateurs littéraux (chaînes littérales et noms de fonctions externes) et les méthodes n'en ayant pas besoin telles que PSS et PSSO.

Bancs d'essai Notre évaluation systématique est séparée en deux bancs d'essai. Le premier banc d'essai consiste en un jeu de données limité et permet une évaluation préliminaire de toutes les méthodes. Le second banc d'essai contient plusieurs jeux de données contenant des centaines de milliers de programmes, et ce banc d'essai ne peut être utilisé qu'avec des méthodes rapides.

Champs de test Pour rappel, la recherche de clones consiste à retrouver dans un dépôt de programme D, un clone d'un programme cible P. La recherche est un succès si le programme le plus similaire de P dans le dépôt D, selon une métrique de similarité M, est un clone. Au lieu de placer chaque programme dans un unique dépôt, nous utilisons différents champs de test pour obtenir une évaluation plus fine. Un *champ de test* (C, D) est composé de l'ensemble C des cibles et de l'ensemble D des programmes dans le dépôt. Par exemple, le champ de test (-O0, -O3) consiste en (i) l'ensemble des programmes cibles, ici chaque programme compilé avec -O0, et (ii) un dépôt contenant chaque programme compilé avec -O3. Par abus de langage, nous notons -O0 comme à la fois le niveau d'optimisation -O0 et l' ensemble des programmes compilés avec -O0.

Scénarios En mesurant le succès au niveau des champs de tests, nous pouvons étudier des scénarios plus ou moins difficiles. Dans un scénario difficile, le dépôt contient des programmes compilés avec le même niveau d'optimisation que les cibles. Or, partager le même niveau d'optimisation risque d'induire une plus grande similarité et c'est alors plus difficile de trouver un clone qui aurait un niveau d'optimisation différent de la cible. Les procédures de recherche de clones font face à de tels scénarios, ce qui explique pourquoi la robustesse est une propriété clé.

Métrique La recherche de clones de programmes est une tâche de *recherche d'information*. Les métriques d'évaluation usuelles sont la précision et le rappel. Nous utilisons la métrique décrite dans l'article Asm2Vec [42], c.a.d. la précision à la première position ou $Precision@1$. La fonction $Precision@1(M, P, D)$ est égale à 1 si et seulement si un clone de la cible P est le programme le plus similaire dans le dépôt D, tel que mesuré par M. Soit $Score(M, T)$, le *score* de précision d'une métrique de similarité M sur l'ensemble des champs de test T. Par symétrie, nous inversons le rôle des dépôts et des ensembles de cibles. Nous définissons donc $Score(M, T) :=$

$$\sum_{(C,D)\in T} \sum_{P\in C} \frac{Precision@1(M, P, D \setminus \{P\})}{2 \times |C| \times |T|} + \sum_{(C,D)\in T} \sum_{P\in D} \frac{Precision@1(M, P, C \setminus \{P\})}{2 \times |D| \times |T|}$$

Implémentation Le désassemblage est réalisé par IDA Pro v7.5 avec un script de la plateforme d'analyse Kam1n0 [17]. Les désassembleurs standard tels que IDA ou Ghidra récupèrent généralement une approximation décente du graphe d'appels, pourvu que l'utilisation des pointeurs de fonctions soit légère. Bien que l'usage intensif des fonctionnalités du langage ou de pointeurs de fonctions puisse en principe poser problème, nos expériences montrent que cela ne semble pas se produire fréquemment en pratique. L'étude approfondie de Pang et al. [117] sur l'état de l'art du désassemblage donne plus de détails sur les capacités et les faiblesses de ces outils. Nos expériences sont réalisées avec deux CPU ayant une fréquence de 2.10 GHz et 20 cœurs par CPU. Tous les temps d'exécution rapportés dans les Chapitres 3, 4 et 5 sont équivalents à des temps d'exécution en utilisant un seul cœur.

TABLE 4.2 – Catalogue des paquets du jeu de données Basique.

Projet	V0	V1	V2	V3
Coreutils	5.93	6.4	7.6	8.30
Binutils	2.25	2.27	2.31	2.35
Findutils	4.233	4.41	4.6	4.7.0
Diffutils	3.1	3.3	3.4	3.6
Bash	4.2	4.3	4.4	5.0
Codeblocks	13.12	16.01	17.12	20.03
Dia	0,94	0,95	0,96	0,97
Geany	1.23	1.27	1.32	1.36
Git	1.9.1	2.7.4	2.17.0	2.25.1
Graphviz	2.36.0	2.38.0	2.40.1	2.42.4
Libsdl2	2.0.10	2.0.14	2.0.2	2.0.8
Lua	5.1.5	5.2.4	5.3.3	5.4.2
Make	3.82	4.1	4.2	4.3
Openssh	6.6p1	7.2p2	7.6p1	8.2p1
Openssl	1.0.1f	1.0.2g	1.1.0g	1.1.1f
Perl	5.18.2	5.22.1	5.26.1	5.30.0
Ruby	1.9.1	2.0.0	2.5	3.0
Subversion	1.14.1	1.8.8	1.9.3	1.9.7
Vlc	2.1.2	2.2.2	3.0.12	3.0.1

17. `https://github.com/McGill-DMaS/Kam1n0-Community`

4.4 Évaluation préliminaire

Les méthodes les plus lentes ne peuvent être évaluées que sur de petits dépôts. Ainsi, l'évaluation préliminaire va tâcher de discerner parmi l'éventail des méthodes les plus rapides, précises et robustes sur un jeu de données contenant environ mille programmes pour le système d'exploitation Linux compilé avec GCC. Les méthodes utilisant des identificateurs littéraux sont séparées durant l'analyse des résultats étant donné la contrainte supplémentaire liée à leur présence et la précision de cette approche. Quatre versions d'une centaine de codes sources de seize paquets logiciels ont été rassemblées. L'étude préliminaire permet alors d'étudier les clones dont c'est la version du code source et non la chaîne de compilation qui change.

4.4.1 Basique, le jeu de données initial

Nous utilisons le code source de plusieurs paquets en 4 versions. La Table 4.2 présente pour chaque paquet le numéro de la version que nous avons utilisé. Chaque programme est attribué un *niveau de version* (c.a.d., V0, V1, V2 ou V3) selon la version de son code source. Les paquets Coreutils, Diffutils et Findutils ont été pris dans le jeu de données de DeepBinDiff [44]. Les paquets venant du jeu de donnée de DeepBinDiff ont été compilés avec GCC v5.4 sur l'architecture x86. Les autres paquets ont été compilés à l'aide de GCC v9.4 sur l'architecture x86.

Nous séparons ces paquets sous la forme de 6 jeux de données, dont l'union est le jeu de données Basique. Cela nous permet de comparer les résultats de plusieurs méthodes selon les paquets, ainsi que les différents niveaux d'optimisation et les différentes versions des codes sources.

- **Coreutils Versions (CV) :** Les quatre versions de 87 codes sources uniques du paquet Coreutils pour un total de 348 programmes. Chaque programme est compilé avec le niveau d'optimisation -O3 ;
- **Coreutils Optimisations (CO) :** Nous extrayons du paquet Coreutils un total de 104 codes sources uniques. Chaque code source est compilé avec quatre niveaux d'optimisation différents (-O0, -O1, -O2, et -O3) ;
- **Utils Versions (UV) :** D'une part, nous sélectionnons les quatre versions de 15 codes sources uniques du paquet Binutils. Chaque programme est compilé avec le niveau d'optimisation -O2. D'autre part, nous extrayons quatre versions de 4 codes sources du paquet Diffutils et quatre versions de 3 codes sources du paquet Findutils. Chaque programme est compilé avec le niveau d'optimisation -O3 ;
- **Utils Optimisations (UO) :** Nous obtenons 88 programmes en compilant les versions les plus récentes des 22 codes sources uniques précédents avec 4 niveaux d'optimisation ;
- **Big Versions (BV) :** À partir de tous nos paquets, nous sélectionnons quatre versions différentes de 21 codes sources uniques pour un total de 84 programmes. Les programmes sont compilés avec le niveau d'optimisation par défaut de chaque paquet ;
- **Big Options (BO) :** Avec les versions les plus récentes des 21 codes sources uniques précédents, nous obtenons 84 programmes en les compilant avec 4 niveaux d'optimisation.

TABLE 4.3 – Caractéristiques du jeu de données Basique.

Jeu de données	CV	CO	UV	UO	BV	BO
Taille (Mo)	18	22	82	91	114	97
Nombre de paquets	1	1	3	3	17	17
Nombre de code source unique	87	104	22	22	21	21
Nombre d'options par code source	1	4	1	4	1	4
Nombre de versions par code source	4	1	4	1	4	1
Nombre de programmes	348	416	88	88	84	84
Taille des dépôt en scénario I	87	104	22	22	21	21
Taille des dépôt en scénario II	173	207	43	43	41	41
Nombre moyen de fonctions	164	219	1222	1456	2160	2930
Nombre moyen de blocs de bases d'une fonction	9	7	19	18	12	11

La Table 4.3 présente les caractéristiques de chaque jeu de données.

Les jeux de données CO et CV contiennent des centaines de petits programmes qui sont difficiles à différencier,c car ils partagent une même origine et des fonctionnalités. À contrario, les jeux de données BO et BV contiennent moins d'une centaine de programmes provenant de différents paquets. En moyenne, un programme de CV contient 164 fonctions, alors qu'un programme de BO contient 2 930 fonctions. Selon les scénarios (voir Section 4.4.2), il est précisé le nombre de programmes dans les dépôts. Au total, le jeu de données complet contient 950 programmes différents, obtenus à partir de 277 codes sources originaux, pour un total de 811 275 fonctions.

4.4.2 Scénarios

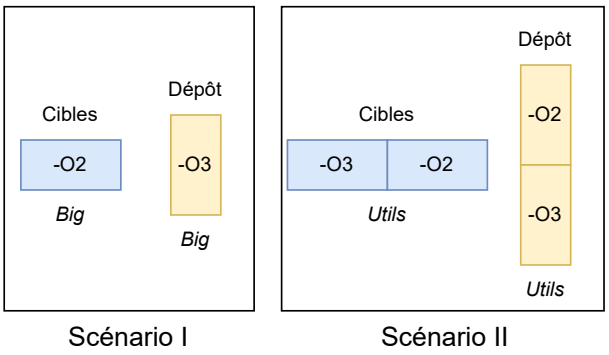


FIGURE 4.1 – Exemples de champs de test pour les deux scénarios.

Nous inventons deux scénarios afin d'évaluer à la fois la précision, la vitesse d'exécution, et la robustesse face aux variations. Dans la Figure 4.1, nous illustrons un exemple de champ de test pour chaque scénario.

Premier scénario Dans le premier scénario, un dépôt contient des programmes ayant le même niveau d'optimisation (ou de version). L'ensemble des cibles correspondant contient des programmes ayant un autre niveau d'optimisation (ou de version). Nous séparons les jeux de données CO, UO et BO en 12 champs de test : (-Oi,-Oj), $\forall\ i, j$ tels que $0 \leq i, j \leq 3$ et $i \neq j$. De même, nous séparons les jeux de données CV, UV et BV en 12 champs de test : (Vi,Vj), $\forall\ i, j$ tels que $0 \leq i, j \leq 3$ et $i \neq j$. Dans la Table 4.3, est précisé le nombre de programmes dans un dépôt dans le premier scénario pour chaque jeu de données.

Le premier scénario est le plus simple, car les similarités entre les optimisations ou versions ne peuvent pas tromper une métrique de similarité. Il permet d'étudier en détail la précision et les temps d'exécution de chacun des douze champs de test.

Second scénario Dans le second scénario, un dépôt contient des programmes ayant deux niveaux d'optimisation (ou de version) différents. L'ensemble des cibles est alors équivalent au dépôt. Nous séparons chaque jeu de données CO, UO, et BO en 6 champs de test : (-Oi $\cup$ -Oj, -Oi $\cup$ -Oj), $\forall\ i, j$ tel que $0 \leq i, j \leq 3$ et $i < j$. De même, nous séparons chaque jeu de données CV, UV, et BV en 6 champs de test : (Vi $\cup$ Vj, Vi $\cup$ Vj), $\forall\ i, j$ tel que $0 \leq i, j \leq 3$ et $i < j$. Dans la Table 4.3, est précisé le nombre de programmes dans un dépôt dans le second scénario pour chaque jeu de données.

Le second scénario est plus complexe. Des similitudes liées au niveau d'optimisation ou à la version peuvent tromper une métrique de similarité. Il permet d'étudier la précision ainsi que la robustesse des métriques de similarité de programme.

4.4.3 RQ1 : Évaluation de la vitesse

La Table 4.4 présente les temps d'exécution dans le premier scénario de chaque méthode sur les jeux de données CV, UV et BV. Ces temps d'exécution incluent les prétraitements spécifiques aux méthodes spectrales. En plus des temps totaux, il est précisé les temps moyens d'une recherche de clones ainsi que les temps maximums d'une recherche.

Les temps sur les programmes du jeu de données CV sont bien plus courts que les temps sur les programmes du jeu de données BV. Par exemple, notre méthode spectrale non optimisée PSS met 22s à effectuer toutes les recherches de clones sur CV, alors qu'elle met 15m sur BV. Cela peut sembler étonnant puisque les dépôts de CV, dans le premier scénario, contiennent 87 programmes alors que ceux de BV contiennent 21 programmes. En fait, le nombre de fonctions des programmes est un facteur clé, et les programmes de CV contiennent en moyenne 164 fonctions tandis que ceux de BV contiennent en moyenne 2160 fonctions. Notre prétraitement étant quadratique par rapport au nombre de fonctions, c'est ce facteur qui prédomine, par rapport aux calculs de similarités qui sont linéaires.

Les méthodes dont le temps de chaque calcul de similarité est plus que linéaire par rapport au nombre de fonctions sont encore plus lentes. Par exemple, la méthode par vectorisation de fonctions SAFE met déjà 9h sur CV, et 175h sur BV. En effet, la métrique de similarité F utilisée

TABLE 4.4 – (RQ1) Temps d'exécution dans le scénario I avec changement de version. Décomposé en : Temps totaux, temps moyen d'une recherche (temps maximum d'une recherche).

Jeu de données	CV	UV	BV
B_{size}	2s, 0,0s (0s)	0s, 0,0s (0s)	0s, 0,0s (0s)
D_{size}	2s, 0,0s (0s)	0s, 0,0s (0s)	0s, 0,0s (0s)
Shape	12s, 0,01s (0s)	11s, 0,04s (0s)	14s, 0,06s (0s)
ASCG	16s, 0,02s (0s)	2m26s, 0,55s (1s)	16m, 3,89s (30s)
ASCFG	4h14m, 14,61s (2m41s)	25h6m, 5m42s (14m)	28h49m, 6m52s (26m)
GED-0	50m, 2,92s (10s)	4h13m, 57,53s (1m45s)	20h23m, 4m51s (17m)
Asm2Vec	1h50m, 6,33s (20s)	14h16m, 3m15s (6m41s)	34h53m, 8m18s (37m)
Gemini	33m, 1,91s (8s)	10h50m, 2m28s (5m15s)	25h23m, 6m3s (24m)
SAFE	9h6m, 31,38s (1m35s)	60h, 13m (28m)	175h, 41m (2h56m)
MutantX-S	2s, 0,0s (0s)	0s, 0,0s (0s)	0s, 0,0s (0s)
PSS	22s, 0,02s (0s)	2m41s, 0,61s (2s)	15m, 3,81s (30s)
PSSO	45s, 0,04s (0s)	2m53s, 0,66s (2s)	3m34s, 0,85s (4s)
GED-Labels	45m, 2,6s (9s)	3h28m, 47,32s (1m33s)	12h28m, 2m58s (14m)
SMIT	2h56m, 10,12s (43s)	75h, 17m (48m)	683h, 2h42m (26h29m)
ISO	0s, 0,0s (0s)	0s, 0,0s (0s)	0s, 0,0s (0s)
CGC	1h49m, 6,27s (20s)	13h17m, 3m1s (6m46s)	49h, 11m (50m)
αDiff	16h1m, 55,25s (2m33s)	57h, 12m (25m)	161h, 38m (2h54m)
LibDX	3s, 0,0s (0s)	17s, 0,06s (0s)	6s, 0,02s (0s)
LibDB	6m41s, 0,38s (2s)	4h42m, 1m4s (2m50s)	1h33m, 22,2s (3m0s)
StringSet	6s, 0,01s (0s)	7s, 0,03s (0s)	5s, 0,02s (0s)
FunctionSet	1s, 0,0s (0s)	0s, 0,0s (0s)	0s, 0,0s (0s)

par SAFE a un temps quadratique par rapport au nombre de fonctions, comme expliqué dans la Section 3.4.2. À l'inverse, les méthodes sans prétraitement dont le temps de calcul de similarité est linéaire (ou moins que linéaire) n'ont pas cette évolution. Nous remarquons bien que, par exemple, StringSet met 6s sur CV et 5s sur BV.

La Table 4.5 présente les temps d'exécution dans le premier scénario de chaque méthode sur les jeux de données CO, UO et BO. Ces jeux de données contiennent des programmes compilés avec différents niveaux d'optimisation, au lieu des niveaux -O2 ou -O3. Puisque les niveaux d'optimisations élevés réduisent le nombre de fonctions, un programme de BV contient en moyenne 2160 fonctions alors qu'un programme de BO contient 2930 fonctions. Ce changement a surtout un impact sur le temps de prétraitement de PSS qui est de 54m sur BO, alors que celui de PSSO reste en dessous de 5m. En revanche, les nombres de programmes dans les dépôts sont à peu près les mêmes pour CO, UO et BO que pour respectivement CV, UV et BV.

TABLE 4.5 – (RQ1) Temps d'exécution dans le scénario I avec changement d'optimisation. Décomposé en : Temps totaux, temps moyen d'une recherche (temps maximum d'une recherche).

Jeu de données	CO	UO	BO
B_{size}	3s, 0,0s (0s)	0s, 0,0s (0s)	0s, 0,0s (0s)
D_{size}	3s, 0,0s (0s)	0s, 0,0s (0s)	0s, 0,0s (0s)
Shape	20s, 0,02s (0s)	10s, 0,04s (0s)	15s, 0,06s (0s)
ASCG	24s, 0,02s (0s)	3m45s, 0,85s (5s)	55m, 13,16s (10m)
ASCFG	4h35m, 13,22s (2m18s)	22h31m, 5m7s (13m)	42h, 10m (36m)
GED-0	2h14m, 6,48s (25s)	6h29m, 1m28s (3m14s)	47h, 11m (1h25m)
Asm2Vec	5h58m, 17,23s (58s)	20h24m, 4m38s (11m)	63h, 15m (2h7m)
Gemini	2h12m, 6,36s (28s)	16h3m, 3m39s (10m)	47h, 11m (1h28m)
SAFE	26h23m, 1m16s (4m22s)	86h, 19m (50m)	297h, 1h10m (8h39m)
MutantX-S	2s, 0,0s (0s)	0s, 0,0s (0s)	0s, 0,0s (0s)
PSS	30s, 0,02s (0s)	4m2s, 0,92s (5s)	54m, 13,03s (10m)
PSSO	1m2s, 0,05s (0s)	2m51s, 0,65s (2s)	4m5s, 0,97s (7s)
GED-Labels	1h57m, 5,66s (22s)	4h57m, 1m8s (2m51s)	23h20m, 5m34s (59m)
SMIT	9h59m, 28,8s (3m58s)	285h, 1h4m (7h1m)	2577h, 10h13m (351h)
ISO	0s, 0,0s (0s)	0s, 0,0s (0s)	0s, 0,0s (0s)
CGC	4h9m, 12s (40s)	18h18m, 4m10s (10m)	84h, 20m (2h7m)
αDiff	40h, 1m57s (5m54s)	79h, 18m (44m)	287h, 1h8m (9h2m)
LibDX	8s, 0,01s (0s)	23s, 0,09s (0s)	7s, 0,03s (0s)
LibDB	41m, 1,98s (9s)	6h33m, 1m30s (4m48s)	2h51m, 40,73s (11m)
StringSet	8s, 0,01s (0s)	7s, 0,03s (0s)	5s, 0,02s (0s)
FunctionSet	1s, 0,0s (0s)	0s, 0,0s (0s)	0s, 0,0s (0s)

Conclusion (RQ1) Nos méthodes spectrales n'ont pas la rapidité de MutantX-S, qui prend un total de quatre secondes pour effectuer toutes les recherches de clones. Ni celle des méthodes de bases ou des méthodes utilisant fortement les identificateurs littéraux (LibDX, StringSet, et FunctionSet) qui mettent moins d'une minute. Néanmoins, nos méthodes spectrales sont bien plus rapides que les méthodes utilisant la distance d'édition de graphes. La plus rapide de ces méthodes est GED-Labels, et elle met plus de 46h. Pire, sa recherche de clones la plus longue met plus de temps à se terminer que l'ensemble des recherches de notre méthode spectrale optimisée. Nous remarquons que les longs calculs de distance d'édition de graphe de SMIT en font la méthode la plus lente avec plus de 3600h. De même, nos méthodes spectrales sont plus rapides que les méthodes par vectorisation de fonctions, dont la plus rapide est Gemini et demande plus de 80h pour rechercher les clones.

TABLE 4.6 – (RQ2) Scores dans le scénario I avec un changement de version des codes sources.

Champ de test	(V0,V1)	(V0,V2)	(V0,V3)	(V1,V2)	(V1,V3)	(V2,V3)
B_{size}	0,35	0,22	0,14	0,32	0,22	0,32
D_{size}	0,28	0,18	0,13	0,35	0,25	0,36
Shape	0,41	0,28	0,19	0,51	0,34	0,43
ASCG	0,63	0,40	0,29	0,54	0,35	0,42
ASCFG	0,27	0,09	0,06	0,07	0,06	0,08
GED-0	0,66	0,38	0,34	0,56	0,44	0,50
Asm2Vec	0,83	0,68	0,60	0,75	0,63	0,71
Gemini	0,78	0,65	0,56	0,73	0,61	0,65
SAFE	0,91	0,73	0,59	0,82	0,66	0,71
MutantX-S	0,80	0,59	0,46	0,72	0,54	0,69
PSS	0,76	0,51	0,35	0,66	0,43	0,55
PSSO	0,74	0,51	0,34	0,64	0,43	0,54
GED-Labels	0,72	0,45	0,31	0,64	0,45	0,55
SMIT	0,11	0,06	0,06	0,12	0,09	0,11
ISO	0,02	0,03	0,02	0,05	0,06	0,02
CGC	0,47	0,42	0,30	0,51	0,39	0,48
αDiff	0,78	0,61	0,50	0,66	0,52	0,58
LibDX	0,71	0,83	0,69	0,81	0,73	0,74
LibDB	0,91	0,86	0,77	0,89	0,79	0,85
StringSet	0,94	0,93	0,93	0,96	0,94	0,93
FunctionSet	0,91	0,89	0,87	0,92	0,88	0,91

4.4.4 RQ2 : Évaluation de la précision

La Table 4.6 présente les scores de précision des différentes méthodes dans le premier scénario et dans le cas de changement de versions. Pour chaque champ de test, c'est la moyenne des scores pour les trois jeux de données Coreutils, Utils et Big qui est indiquée. Le champ de test (V0,V3) est le plus difficile puisque les versions des programmes sont les plus éloignées et il est ardu de faire correspondre la cible en version V0 avec un clone en version V3 dans le répertoire. Sur celui-ci, les méthodes par vectorisation telles que Asm2Vec, SAFE et Gemini ont des scores importants de $0,60$, $0,53$ et $0,56$ respectivement. La méthode par N-gramme MutantX-S est derrière avec $0,46$, et nos méthodes spectrales PSS et PSSO atteignent $0,35$ et $0,34$. Des méthodes avec des identificateurs tels que StringSet, FunctionSet, LibDB et LibDX sont plus précises et atteignent respectivement $0,93$, $0,87$, $0,77$ et $0,69$. À contrario, certaines méthodes utilisant pourtant les identificateurs comme GED-Labels, et la méthode de couplage CGC ont des scores de $0,31$ et $0,30$, elles ne sont pas plus précises que nos méthodes spectrales. Nous notons également que αDiff qui combine la vectorisation et l'utilisation des noms des fonctions externes a un score de

TABLE 4.7 – (RQ2) Scores dans le scénario I avec un changement de niveau d'optimisation.

Champ de test	(O0,O1)	(O0,O2)	(O0,O3)	(O1,O2)	(O1,O3)	(O2,O3)
B_{size}	0,19	0,21	0,31	0,54	0,32	0,30
D_{size}	0,24	0,19	0,31	0,53	0,33	0,28
Shape	0,22	0,24	0,23	0,48	0,48	0,50
ASCG	0,21	0,14	0,12	0,62	0,45	0,52
ASCFG	0,10	0,07	0,06	0,14	0,05	0,08
GED-0	0,32	0,32	0,31	0,65	0,56	0,64
Asm2Vec	0,36	0,33	0,35	0,80	0,71	0,88
Gemini	0,16	0,14	0,13	0,80	0,62	0,77
SAFE	0,26	0,22	0,19	0,72	0,63	0,90
MutantX-S	0,22	0,22	0,22	0,39	0,33	0,73
PSS	0,32	0,27	0,21	0,75	0,63	0,67
PSSO	0,29	0,27	0,22	0,80	0,62	0,67
GED-Labels	0,36	0,27	0,23	0,57	0,56	0,75
SMIT	0,08	0,09	0,11	0,03	0,05	0,05
ISO	0,02	0,02	0,04	0,05	0,06	0,02
CGC	0,16	0,19	0,21	0,63	0,36	0,58
αDiff	0,24	0,24	0,22	0,75	0,58	0,86
LibDX	0,91	0,92	0,93	0,59	0,70	0,48
LibDB	0,60	0,53	0,47	0,86	0,81	0,89
StringSet	0,96	0,94	0,96	0,94	0,96	0,97
FunctionSet	0,89	0,90	0,89	0,92	0,94	0,94

$0,50$ ce qui est derrière les méthodes par vectorisation plus classique.

Comme attendu, les scores sont globalement plus élevés pour les méthodes n'utilisant pas d'identificateurs littéraux sur les autres champs de test. Par exemple, sur (V0,V1), SAFE atteint un score de $0,91$. Néanmoins, l'ordre des meilleures classes de méthodes est le même, d'abord les méthodes par vectorisation comme SAFE puis MutantX-S $(0,80)$ et ensuite nos méthodes spectrales PSS $(0,76)$ et PSSO $(0,74)$. Sur le champ de test (V0,V1), les méthodes se basant directement sur les identificateurs littéraux ont des scores similaires par rapport à leurs scores sur (V0,V3). Par exemple, StringSet a un score de $0,94$ au lieu de $0,93$. Cela atteste qu'entre des versions de code source, les identificateurs littéraux évoluent très peu.

La Table 4.7 présente les scores de précision des différentes méthodes du premier scénario dans le cas de changement de niveau d'optimisation. Sans utiliser d'identificateurs littéraux, nous remarquons en général des scores plus élevés sur le champ de test (O2,O3). Ces scores élevés peuvent être expliqués par le peu de différence entre un programme optimisé au niveau 2 et au niveau 3. Les méthodes par vectorisation sont les meilleures avec par exemple SAFE $(0,90)$,

puis MutantX-S (0, 73) et PSS et PSSO (0, 67 pour les deux). L'ordre des classes de méthodes les plus précises est proche de celui énoncé précédemment. Nous remarquons cependant que nos méthodes spectrales dépassent ou sont proches de MutantX-S sur les autres champs de test, avec 0, 75 contre 0, 39 sur (O1,O2) par exemple. En utilisant des identificateurs littéraux, StringSet est systématiquement la méthode la plus précise, avec 0, 96 sur le champ de test (O0,O3), viennent ensuite LibDX (0, 93) et FunctionSet (0, 89). Nous remarquons que LibDB n'atteint qu'un score de 0, 47. Cela traduit l'affaiblissement de sa partie utilisant la vectorisation de fonctions par le changement important d'optimisation.

Conclusion La précision dans le premier scénario est la meilleure pour les méthodes utilisant les identificateurs littéraux, notamment nos deux heuristiques StringSet et FunctionSet, puis LibDX et LibDX. Autrement, les méthodes par vectorisations sont les plus précises. Notamment SAFE pour repérer des versions différentes et Asm2Vec pour des niveaux d'optimisations différents. Viennent ensuite nos méthodes spectrales et enfin la méthode de N-gramme MutantX-S. Elle est meilleure que nos méthodes pour repérer des versions différentes, mais moins précises face à des niveaux d'optimisations proches.

Second scénario La Table 4.8 présente les scores de précision des méthodes dans le second scénario. Pour chaque jeu de données, la moyenne des scores de chaque champ de test est indiquée. Nous remarquons premièrement la plus grande difficulté des recherches de clones dans les jeux de données CV et CO. Par exemple, PSS atteint un score de 0, 16 sur CV et 0, 10 sur CO. Cela est sans doute dû d'une part à la taille plus importante des dépôts, mais aussi au fait que les programmes de Coreutils sont très similaires entre eux en plus d'être petit. Comme attendu, PSS atteint des scores élevés de 0, 75et 0, 65 sur BV et BO. Il est facile de différencier les larges programmes de BV et BO qui viennent de paquets ne partageant pas de code source. Deuxièmement, le classement des meilleures méthodes en termes de précision est différent dans le second scénario par rapport au premier. Les méthodes par vectorisation de fonctions s'effondrent quand on ajoute d'autres niveaux d'optimisation, Gemini n'obtient que 0, 05 sur UO alors que PSS a 0, 28. De même, la méthode par N-gramme MutantX-S n'obtient que 0, 13. Dans la question suivante, nous montrons que cela est dû à un manque de robustesse face aux changements de niveau d'optimisation. Nous remarquons de même qu'une méthode avec identificateur comme StringSet est proche de PSS sur les jeux de données CV (0, 17), CO (0, 12), et UV (0, 36).

Conclusion (RQ2) La précision est globalement meilleure pour certaines méthodes utilisant les identificateurs littéraux, comme StringSet, FunctionSet, LibDX et LibDX. Autrement, PSS est globalement la plus précise puisqu'elle se comporte bien dans le cas général. Les méthodes par vectorisation de fonctions sont meilleures dans le premier scénario. De même, la méthode MutantX-S a des soucis lors d'un changement de niveau d'optimisation. La méthode GED-0 et la méthode spectrale standard ASCG ont une précision plus faible, démontrant l'intérêt de PSS.

TABLE 4.8 – (RQ2) Scores dans le second scénario.

Jeu de données	CV	CO	UV	UO	BV	BO	Moyenne
B_{size}	0,05	0,04	0,30	0,25	0,17	0,34	0,19
D_{size}	0,04	0,02	0,27	0,24	0,27	0,40	0,21
Shape	0,03	0,01	0,27	0,12	0,51	0,67	0,27
ASCG	0,10	0,03	0,39	0,30	0,58	0,56	0,32
ASCFG	0,08	0,03	0,06	0,04	0,11	0,10	0,07
GED-0	0,11	0,02	0,35	0,28	0,67	0,77	0,37
Asm2Vec	0,01	≈ 0	0,03	0,02	0,45	0,37	0,15
Gemini	0,15	0,01	0,23	0,05	0,83	0,46	0,29
SAFE	0,08	0,01	0,25	0,06	0,81	0,42	0,27
MutantX-S	0,14	0,01	0,37	0,13	0,79	0,40	0,31
PSS	0,16	0,10	0,36	0,28	0,75	0,65	0,38
PSSO	0,17	0,11	0,36	0,28	0,73	0,65	0,38
GED-Labels	0,16	0,19	0,40	0,23	0,62	0,65	0,37
SMIT	0,02	≈ 0	0,08	0,07	0,14	0,10	0,07
ISO	0,01	≈ 0	0,02	0,04	0,02	0,02	0,01
CGC	0,05	0,04	0,39	0,24	0,51	0,44	0,28
αDiff	0,06	≈ 0	0,32	0,16	0,74	0,43	0,28
LibDX	0,56	0,69	0,55	0,70	0,82	0,85	0,69
LibDB	0,19	0,07	0,41	0,20	0,88	0,79	0,42
StringSet	0,17	0,12	0,36	0,42	0,88	0,90	0,48
FunctionSet	0,40	0,51	0,55	0,52	0,87	0,89	0,62

4.4.5 RQ3 : Évaluation de la robustesse

Une difficulté usuelle de la détection de similarité est qu'une méthode pourrait considérer deux programmes comme similaires en se basant sur des propriétés communes secondaires (p. ex., l'architecture, le compilateur utilisé ou le niveau d'optimisation) considérées comme hors sujet du point de vue de la recherche de clones.

Nous considérons la robustesse d'une métrique de similarité comme sa résistance à l'impact de propriétés telles que le niveau d'optimisation ou le niveau de version. Idéalement, parce que le niveau d'optimisation n'influence pas la sémantique, le partage du niveau d'optimisation ne devrait pas contribuer à un plus haut indice de similarité.

Méthode Nous évaluons la sensibilité des différentes approches à ce type de biais en calculant des corrélations rang-bisérial entre (a) le partage d'un niveau d'optimisation ou de version et (b) la similarité des classements dans les nouvelles recherches de clones.

Durant chaque recherche de clones, nous classons les programmes dans le dépôt des plus similaires aux moins similaires, selon une métrique de similarité. Dans le second scénario, nous pouvons diviser ce classement en une liste A contenant des programmes avec le même niveau d'optimisation (ou de version) et une liste B contenant des programmes avec un niveau d'optimisation (ou de version) différent. Notre hypothèse statistique H est que les programmes sont plus proches des autres programmes partageant les mêmes niveaux d'optimisation (ou de version). Statistiquement parlant, cela signifie que les programmes de la liste A ont des rangs plus élevés que les programmes de la liste B. Nous pouvons donc calculer une corrélation rang-bisérial entre la variable dichotomique (l'appartenance à la liste A) et la variable ordinale (le rang).

Nous effectuons ce processus pour chaque recherche de clones dans le deuxième scénario et recueillons plusieurs corrélations pour chaque méthode. Lorsque la corrélation est positive, le partage d'une propriété est associé à un indice de similarité plus élevé. Et lorsque la corrélation est négative, le partage d'une propriété est associé à un indice de similarité plus bas. Une valeur proche de zéro est préférable pour une métrique de similarité.

Résultats Nous écrivons les corrélations pour chaque jeu de données dans la Table 4.9. Sur les jeux de données CV et CO, les méthodes par vectorisation de fonctions Asm2Vec, Gemini et SAFE ont une très forte corrélation pour notre hypothèse. Par exemple, les recherches de clones effectuées avec Asm2Vec atteignent une corrélation moyenne de $0,99$ sur CV et 1 sur CO. Nous pensons qu'Asm2Vec est moins robuste en raison de sa phase d'entraînement spécifique. Cela souligne la nécessité d'entraîner les méthodes à base d'apprentissage sur un panel de programme. αDiff fournit une corrélation modérément élevée de $0,60$ sur CV, mais une très forte corrélation de $0,93$ sur CO. Nous notons que la méthode FunctionSet a une corrélation modérée de $0,39$ sur CV et $0,37$ sur CO. Alors que la méthode par N-gramme MutantX-S a une corrélation modérément élevée de $0,39$ sur CV, elle a une corrélation de $0,63$ sur CO. Ces hautes corrélations s'expliquent sans doute par le fait que les jeux de données CV et CO contiennent de petits

programmes similaires entre eux. Comme attendu, le partage d'un niveau d'optimisation est un facteur important pour les méthodes par vectorisation de fonction. Les autres méthodes fournissent de faibles corrélations inférieures à $0,35$ pour l'hypothèse. Notamment, notre méthode spectrale PSS a une faible corrélation de $0,06$ sur CV et de $0,13$ sur CO. Sur les jeux de données UV et UO, Asm2Vec fournit des corrélations modérément élevées de notre hypothèse, avec $0,49$ sur UV et $0,65$ sur UO. Cependant, les autres méthodes par vectorisation de fonctions fournissent une corrélation modérée. Par exemple, Gemini a une corrélation de $0,19$ sur UV et de $0,37$ sur UO. Notre méthode spectrale a une faible corrélation de $0,02$ sur UV et $0,09$ sur UO. Nous constatons à nouveau que le partage d'un niveau d'optimisation est un facteur plus important que le partage d'un niveau de version. Sur les jeux de données BV et BO, Asm2Vec fournit une corrélation modérée pour notre hypothèse avec $0,32$ sur BV et $0,45$ sur BO. Cependant, sur ce jeu de données contenant de larges programmes qui ne sont pas liés entre eux, Gemini et SAFE ont de très faibles corrélations. Notre méthode spectrale a de faibles corrélations de $-0,04$ sur BV et $0,02$ sur BO. Les autres méthodes fournissent des corrélations absolues inférieures à $0,11$.

TABLE 4.9 – (RQ3) Corrélations moyennes de l'hypothèse H selon la méthode utilisée.

Jeu de données	CV	CO	UV	UO	BV	BO	Moyenne
B_{size}	0,17	**0,07**	**-0,02**	**0,03**	**-0,03**	**-0,04**	**0,03**
D_{size}	**0,11**	**0,02**	**-0,02**	**0,06**	**-0,04**	**-0,04**	**0,02**
Shape	**0,15**	**0,10**	**-0,03**	**0,06**	**-0,04**	**-0,04**	**0,03**
ASCG	**0,10**	0,19	**-0,01**	**0,08**	**-0,04**	**-0,04**	**0,05**
ASCFG	0,23	0,30	**0,15**	0,17	**-0,02**	**-0,02**	**0,13**
GED-0	0,22	0,25	**-0,02**	**0,05**	**-0,04**	**-0,04**	**0,07**
Asm2Vec	0,99	1	0,49	0,65	0,32	0,45	0,65
Gemini	0,76	0,96	0,19	0,37	**-0,04**	**0,06**	0,38
SAFE	0,81	0,98	0,20	0,38	**-0,04**	**0,11**	0,41
MutantX-S	0,39	0,63	**0,05**	0,28	**-0,04**	**0,08**	0,23
PSS	**0,06**	**0,13**	**0,02**	**0,09**	**-0,04**	**-0,02**	**0,04**
PSSO	**0,07**	**0,12**	**0,02**	**0,09**	**-0,04**	**-0,02**	**0,04**
GED-Labels	0,16	0,21	**-0,01**	**0,08**	**-0,04**	**-0,04**	**0,06**
SMIT	**-0,15**	-0,57	**-0,15**	-0,44	**-0,01**	**-0,07**	-0,23
ISO	$\approx$ **0**	$\approx$ **0**	$\approx$ **0**	**0,03**	$\approx$ **0**	**-0,01**	$\approx$ **0**
CGC	0,19	0,32	**0,08**	**0,07**	**-0,04**	**-0,08**	**0,09**
αDiff	0,60	0,93	0,19	0,33	**-0,04**	**0,11**	0,35
LibDX	0,32	**-0,02**	**-0,05**	-0,17	**-0,05**	**-0,04**	$\approx$ **0**
LibDB	0,54	0,46	0,17	0,21	**-0,04**	**-0,03**	0,22
StringSet	0,73	0,86	0,17	0,31	**-0,04**	0,18	0,37
FunctionSet	0,39	0,37	**0,08**	0,22	**-0,04**	**0,02**	0,17

Corrélation absolu moyenne inférieur à 0,16

Conclusion (RQ3) Une méthode parfaitement robuste n'est pas influencée par les niveaux d'optimisation ou la version du code source et sa corrélation moyenne pour l'hypothèse H serait donc proche de 0. Selon ce critère, les méthodes spectrales sont robustes. PSS offre une corrélation moyenne de $0,04$. De plus, la méthode par distance d'édition de graphe GED-0 est également robuste avec une moyenne de $0,07$. C'est la confirmation de notre idée de départ selon laquelle les méthodes GED sont robustes, et que les méthodes spectrales devraient partager leur robustesse. Les points de comparaison et les méthodes par couplage sont également robustes. Par exemple, CGC obtient une moyenne de $0,09$. Mieux encore, nous remarquons que LibDX, qui utilise des identificateurs littéraux, est parfaitement robuste avec une moyenne proche de 0. Deuxièmement, les autres méthodes sont peu robustes. Par exemple, FunctionSet a une corrélation de $0,17$, LibDB de $0,22$ et StringSet de $0,37$. La méthode par N-gramme MutantX-S a une corrélation de $0,22$ ce qui est modérément faible. Dans l'ensemble, les méthodes par fonction sont les moins robustes puisqu'elles obtiennent des corrélations allant de $0,35$ pour αDiff à $0,65$ pour Asm2Vec.

4.4.6 RQ4 : Impact des composantes de PSS

La méthode PSS peut être pensée comme la combinaison de deux composantes. La première composante est une comparaison des valeurs propres des graphes d'appels de fonctions. La métrique de similarité correspondante est simCG. La seconde composante est une comparaison des nombres d'arêtes des graphes de flot de contrôle (CFG). La métrique de similarité correspondante est simCFG.

Composantes actuelles La Table 4.10 rapporte, suivant les jeux de données, la moyenne dans les deux scénarios des scores de précision des composantes. Nous remarquons que simCG obtient de meilleurs résultats que simCFG sur CV et CO, la différence est de respectivement $0,03$ et $0,08$. Cependant, simCG est meilleur que simCFG sur les autres jeux de données. Par exemple, simCG a une moyenne de $0,51$ sur BO, alors que simCFG obtient $0,36$. PSS obtient de meilleurs résultats que les deux composantes dans les deux scénarios. Par exemple, PSS obtient une moyenne de $0,51$ dans le premier scénario, alors que simCG obtient $0,47$ et simCFG seulement $0,40$.

Table 4.10 – Scores des composantes de PSS.

Jeu de données Composante	CV	CO	UV	UO	BV	BO	Scénario I	Scénario II
simCG	0,09	0,05	***0,30***	***0,26***	***0,49***	***0,51***	***0,47***	***0,37***
simCFG	***0,12***	***0,08***	0,27	0,17	0,44	0,36	0,40	0,31
PSS	**0,15**	**0,12**	**0,31**	0,23	**0,51**	0,46	**0,51**	**0,38**

meilleur composante, **meilleur métrique**

Ajout d'une composante lexicale à PSS D'après nos résultats, les méthodes FunctionSet et StringSet, qui ne font que comparer des ensembles d'identificateurs littéraux, sont précises. Nous examinons la possibilité de combiner PSS et FunctionSet ou StringSet. Nous définissons deux nouvelles métriques : $PSSxFS(\alpha, a, b) := (1-\alpha)FunctionSet(a, b) + \alpha PSS(a, b)$ et $PSSxSS(\alpha, a, b) := (1-\alpha)StringSet(a, b) + \alpha PSS(a, b)$. Les Figures 4.2 et 4.3 décrivent l'évolution du score quand on augmente l'importance de PSS dans les méthodes utilisant les identificateurs littéraux. Dans les deux scénarios, la précision ne fait presque que baisser. Il ne semble donc pas y avoir un intérêt à ajouter une composante lexicale à PSS.

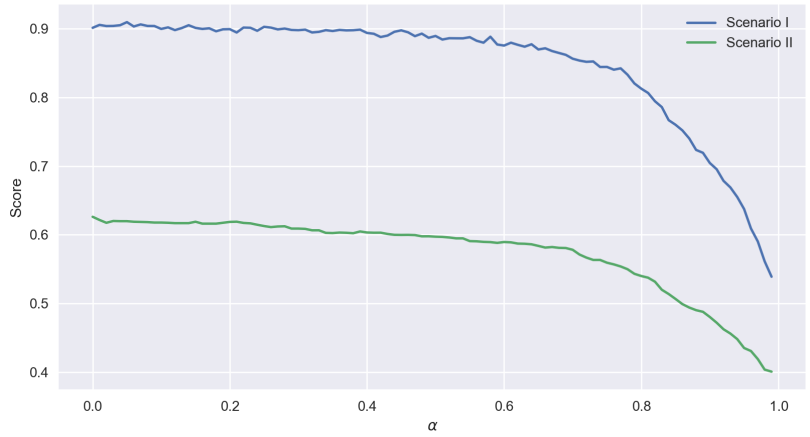


FIGURE 4.2 – Courbe des scores quand α, l'importance de PSS, augmente pour $PSSxFS$.

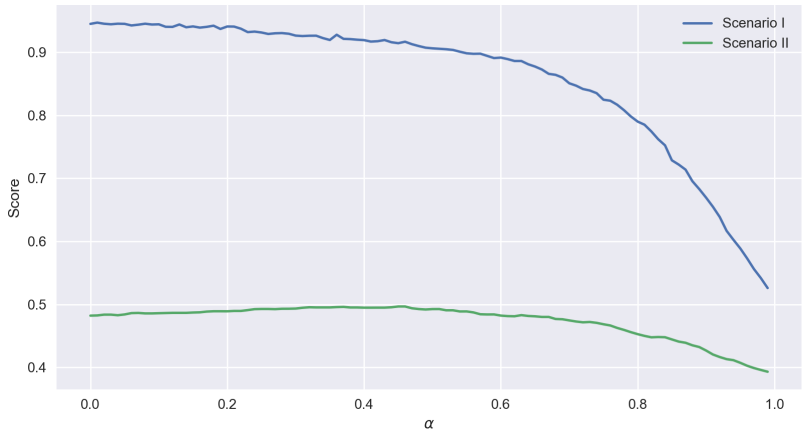


FIGURE 4.3 – Courbe des scores quand α, l'importance de PSS, augmente pour $PSSxSS$.

Conclusion (RQ4) PSS est un bon compromis dans les deux scénarios. D'un côté, la composante simCG est pertinente sur les jeux de données Utils et Big. Mais de l'autre, simCFG est critique sur les jeux de données Coreutils. Nous pouvons émettre l'hypothèse que les nombres d'arêtes des CFG sont plus importants dans le cas des petits programmes. Accessoirement, dans le cas où des identificateurs littéraux seraient présents, les utiliser avec PSS n'apporte rien.

4.4.7 Conclusion de l'étude préliminaire

Pour dresser un bilan de l'étude préliminaire, nous considérons l'inverse des temps totaux dans le premier scénario (Tables 4.4 et 4.5) comme un synonyme de vitesse. De plus, nous considérons que des scores de précision élevés dans les deux scénarios (Tables 4.6, 4.7 et 4.8) induisent une meilleure précision. Enfin, nous considérons qu'une corrélation absolue faible pour une méthode dans la Table 4.9 induit une bonne robustesse. Après l'utilisation de moyennes, nous obtenons trois tableaux de données. La dernière étape est de la transformation des valeurs de ces tableaux en des réels compris entre 0 et 5 via la normalisation par quantiles.

Le bilan illustré sous forme de barres empilées est donné par la Figure 4.4.

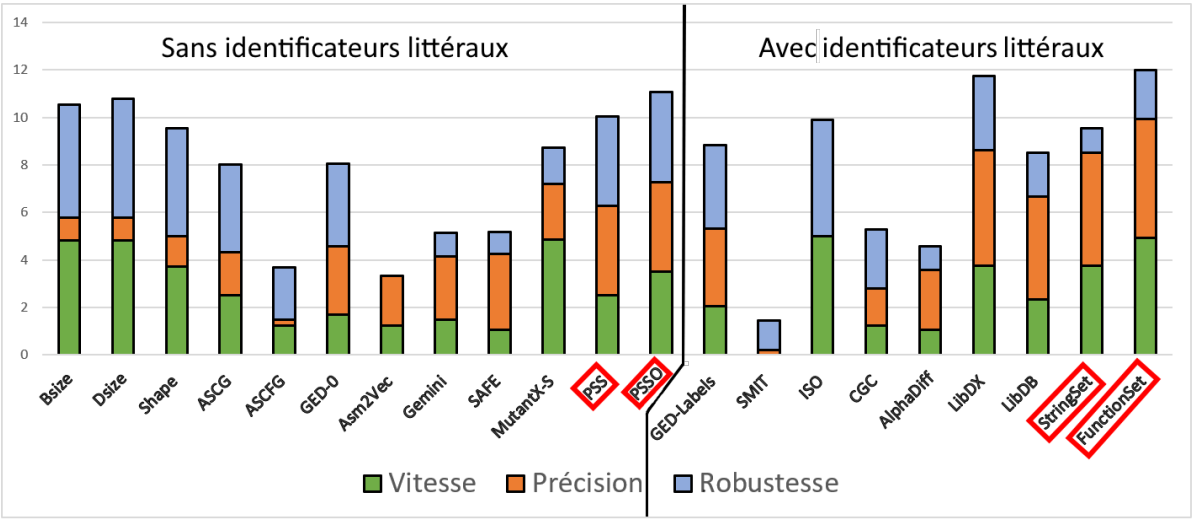


FIGURE 4.4 – Bilan de notre étude préliminaire sur le jeu de données Basique.

Sur ce petit jeu de données de 950 programmes dépouillés de symboles, nos méthodes spectrales PSS et PSSO sont particulièrement robustes et précises, y compris dans un scénario difficile. Elles sont plus précises que les méthodes par distance d'édition de graphe ou par couplage tout en étant bien plus rapide. Nos méthodes spectrales dépassent en tout point les adaptations standard de l'analyse spectrale des graphes.

La méthode par N-gramme MutantX-S est plus rapide, de plus elle est aussi précise que nos méthodes spectrales dans le premier scénario qui ne met pas en jeu la robustesse. De même, les méthodes par vectorisation de fonctions sont très précises dans ce scénario, mais très peu robustes face à l'impact des niveaux d'optimisations. De plus, les méthodes par vectorisation de fonctions sont bien plus lentes que PSS et particulièrement PSSO.

Il faut toutefois noter que sur le jeu de données Basique, certaines méthodes utilisant des identificateurs littéraux comme LibDX, StringSet et FunctionSet sont les plus précises et rapides. Nos deux propositions StringSet et FunctionSet ont l'air d'avoir un avantage de rapidité par rapport à LibDX. Cependant, LibDX semble légèrement plus robuste.

4.5 Mise à l'échelle de la recherche de clones

Nous souhaitons maintenant évaluer le potentiel de PSS en termes de vitesse, de précision, et de robustesse sur de larges dépôts. Alors, nous éliminons les méthodes incapables de réaliser toutes les recherches de clones dans le premier scénario sur le petit jeu de données Basique en moins de 1h30m. Les temps totaux des recherches sont présentés dans la Table 4.11. Les temps d'apprentissage sont eux présentés dans la Table 4.12.

TABLE 4.11 – Temps totaux des recherches dans le premier scénario sur le jeu de données Basique.

B$_{size}$	$\leq$ 1h30m
D$_{size}$	$\leq$ 1h30m
Shape	$\leq$ 1h30m
ASCG	$\leq$ 1h30m
MutantX-S	$\leq$ 1h30m
PSS	$\leq$ 1h30m
PSSO	$\leq$ 1h30m
LibDX	$\leq$ 1h30m
StringSet	$\leq$ 1h30m
FunctionSet	$\leq$ 1h30m

ASCFG	42h
GED-0	81h
GED-L	46h
SMIT	3634h
CGC	171h
Asm2vec	141h
Gemini	102h
SAFE	655h
αDiff	642h
LibDB †	16h

Méthodes rapides sélectionnées pour l'analyses sur des dépôts larges.

Nous constatons des différences significatives entre les méthodes, 10 d'entre elles sont capables de réussir en moins de 1h30, et souvent beaucoup moins, tandis que les dix autres méthodes nécessitent beaucoup plus de temps, de 16h à 3634h. Asm2Vec fait plus de phases d'apprentissages (voir la Section 4.2), d'où son temps d'apprentissage total de 443h. Les temps d'apprentissage pour une époque vont de 21m à 3h, et seraient bien plus importants sur de plus grands dépôts.

Comme déjà mentionné dans la Section précédente, les méthodes par similarité de fonctions, utilisant typiquement de l'apprentissage [42, 101, 106, 141, 154], et les méthodes calculant des distances d'édition de graphe [54,69,129] ne peuvent pas être appliquées sur des jeux de données importants. Par la suite, nous allons considérer uniquement nos méthodes PSS, PSSO, StringSet et FunctionSet, nos trois points de comparaison (B_{size}, D_{size}, Shape), ainsi que les autres outils de l'état de l'art ASCG [54], MutantX-S [68] et LibDX [141].

TABLE 4.12 – Temps d'apprentissage dans le premier scénario sur le jeu de données Basique.

Méthode	Temps total	Temps par époque
Asm2Vec	443h	2h57m
Gemini	17h	21m33s
αDiff	58h	2h56m

4.5.1 Jeux de données

Jeu de données BinKit Afin d'étudier les résultats des méthodes rapides selon les options de compilation, les compilateurs, les architectures et les offuscations, nous réutilisons deux jeux de données de programme Linux provenant de BinKit [85] :

- **Normal :** Depuis 51 paquets logiciels GNU, 235 codes sources uniques ont été extraits. Ils ont été compilés avec 288 chaînes de compilation différentes pour un total de 67 680 programmes d'une taille moyenne de 201 Ko. Cela couvre huit architectures (ARM, x86, MIPS, et MIPSEB, toutes disponibles en 32 et 64 bits), neuf compilateurs (cinq versions de GCC et quatre versions de Clang), ainsi que quatre niveaux d'optimisation de -O0 à -O3 ;
- **Offuscation :** Quatre offuscations sont considérées en utilisant Obfuscator-LLVM [81] comme compilateur. Celles-ci sont (i) la substitution d'instruction (SUB), (ii) le bug de flot de contrôle (BCF), (iii) l'aplatissement du flot de contrôle (FLA), et (iv) la combinaison des trois précédentes. Les mêmes architectures et niveaux d'optimisation que pour Normal sont couverts, pour un total de 30 080 programmes d'une taille moyenne de 514 Ko.

Jeu de données IoT malveillants Grâce à MalwareBazaar [18], nous rassemblons 19 959 micrologiciels malveillants ciblant les objets connectés. La Figure 4.5 présente un histogramme des tailles des fichiers, qui ont une moyenne de 84 Ko et ne dépassent pas un Mo. Ceux-ci ont été soumis entre mars 2020 et mai 2022. Ils couvrent huit architectures surtout ARM, MIPS, Motorola 68040 et SPARC, voir la Figure 4.7. En utilisant les métadonnées disponibles, que ce soit des rapports d'antivirus ou des règles YARA, nous décomposons les données en trois familles de clones : 12 357 Mirai, 5 842 Gafgyt, et 1 760 Tsunami.

Jeu de données Windows Nous assemblons un jeu de données de 84 992 programmes sains fonctionnant sous les systèmes d'exploitation Windows pour l'architecture x86. Cela représente plus de 50 Go de programmes bruts, avec une taille moyenne de 771 Ko. La Figure 4.6 présente un histogramme des tailles des fichiers, nous remarquons que certains programmes ont un poids de plusieurs dizaines de Mo. Ces programmes vont d'une bibliothèque de Windows XP à un programme de Windows 10. Le jeu de donnée contient également 30 621 mises à jour de sécurité des vingt dernières années. À l'exclusion des mises à jour de sécurité, le jeu de données contient plus de 28 000 bibliothèques dynamiques. Les programmes sont divisés selon leur plateforme, par exemple, Windows 7. Par commodité, nous classons 10 717 programmes qui ne sont pas associés à une plateforme spécifique comme faisant partie d'une plateforme commune. La Figure 4.8 présente la distribution des plateformes par une décomposition en rectangle. Nous considérons que deux programmes partageant à la fois (i) un nom de fichier et (ii) une plateforme sont des clones. Au total, 49 443 programmes ont un clone et sont donc des cibles d'une recherche de clones.

18. `https://bazaar.abuse.ch/`

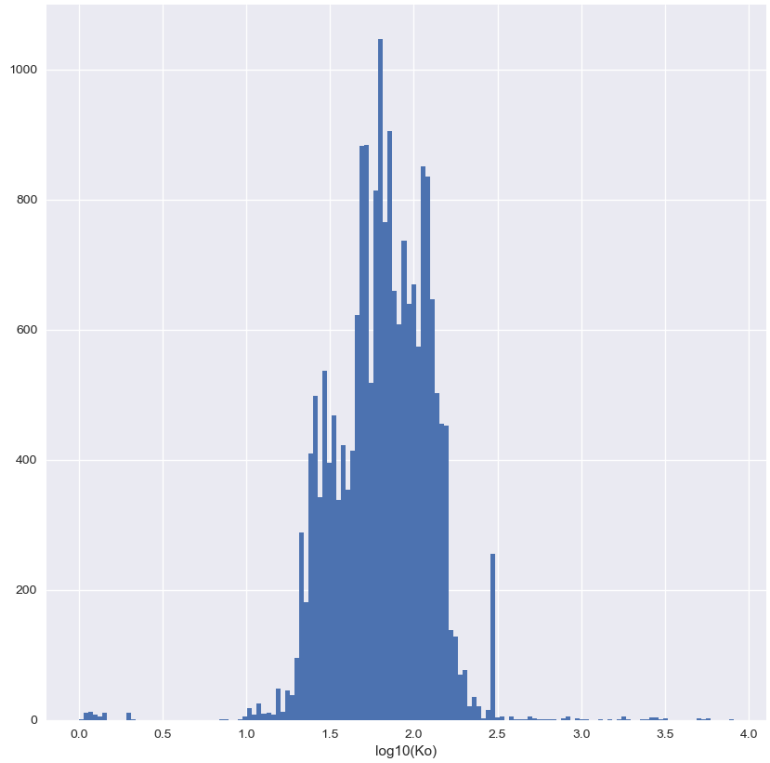


FIGURE 4.5 – Histogramme de la taille en Ko (log10) des programmes IoT malveillants.

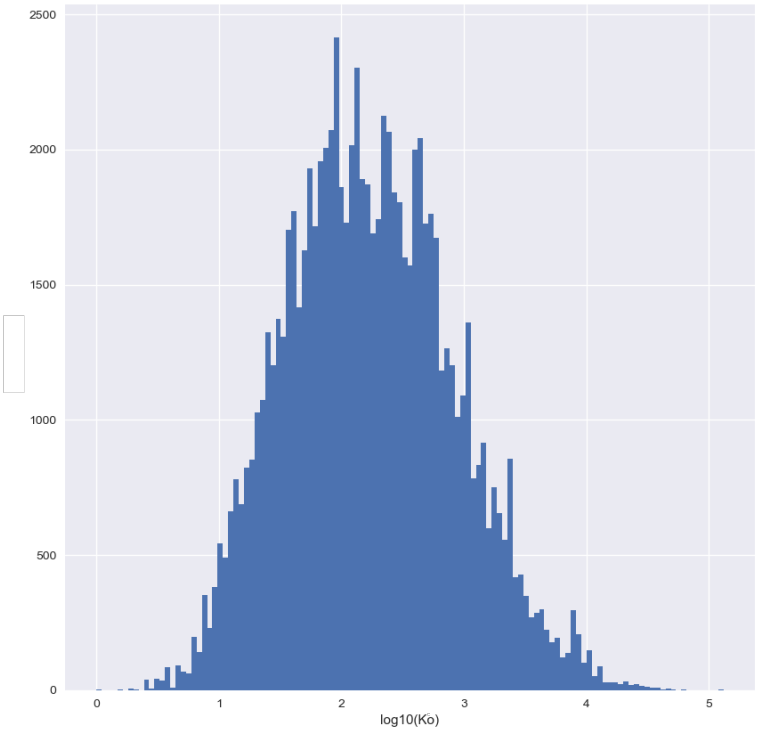


FIGURE 4.6 – Histogramme de la taille en Ko (log10) des programmes Windows.

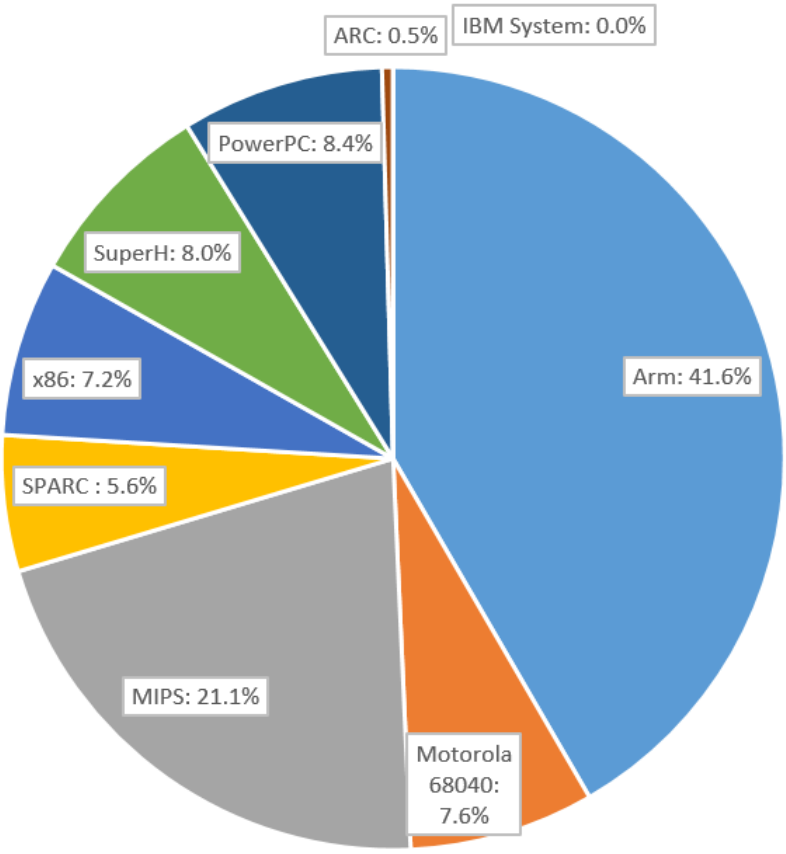


FIGURE 4.7 – Distribution des architectures des programmes IoT malveillants.

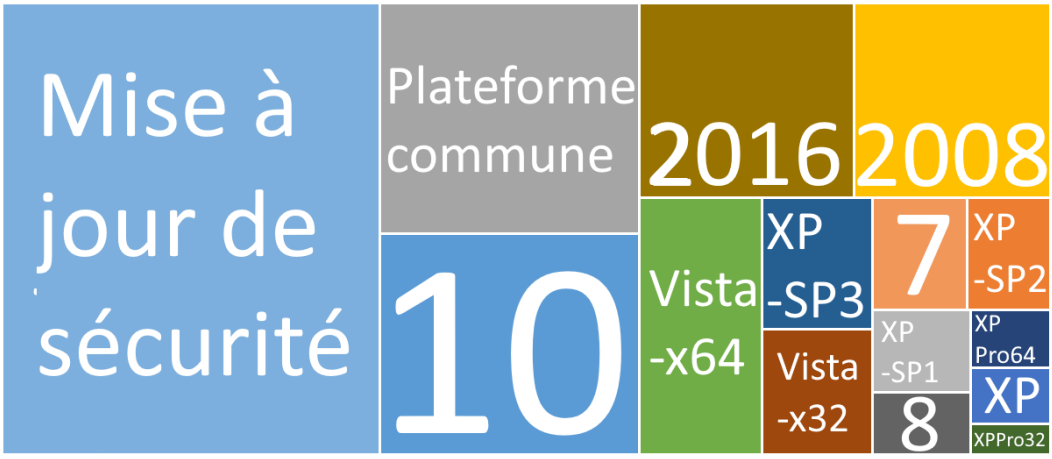


FIGURE 4.8 – Distribution des plateformes des programmes Windows.

4.5.2 Scénarios

Concernant le jeu de données BinKit, nous nous plaçons dans le premier scénario (voir la Section 4.4.2). Ainsi, un dépôt contient des programmes partageant une caractéristique, par exemple ils ont le même niveau d'optimisation. L'ensemble des programmes cibles correspondant ne contient que des programmes avec une caractéristique différente, telle qu'un niveau d'optimisation différent. Nous séparons le jeu de donnée BinKit entre les niveaux d'optimisations, les versions de compilateurs, les compilateurs, les architectures, et les offuscations produisant 54 champs de test.

En revanche, en ce qui concerne les jeux de données IoT et Windows, nous nous plaçons dans le scénario où un unique dépôt contient l'ensemble des programmes.

4.5.3 RQ1 : Évaluation de la vitesse

Nous rapportons dans la Table 4.13 les temps d'exécution et les temps de prétraitement sur les grands jeux de données : BinKit, IoT et Windows.

BinKit Sur le jeu de données BinKit, qui contient 97 760 programmes d'une taille moyenne de 313 Ko, PSS prend 190h, PSSO 116h et MutantX-S est plus lent avec 220h. Parmi les méthodes utilisant des identificateurs littéraux, LibDX et StringSet sont beaucoup plus lentes (resp., 1965h et 542h). En revanche, FunctionSet est rapide (37h).

IoT Sur le jeu de données IoT contenant 19 959 programmes malveillants ciblant les objets connectés, PSS ne prend que 2h9m. Il est plus rapide que MutantX-S (3h34m). Surprenamment, PSSO est un peu plus lent que PSS et prend 2h12m. Parmi les méthodes qui utilisent des identificateurs littéraux, FunctionSet est rapide, avec moins de 8 minutes en total. LibDX prend 7h47m, tandis que StringSet est le plus lent avec 9h21m.

Windows PSS prend 263h sur le jeu de données des programmes Windows. C'est beaucoup plus que MutantX-S (41h) et un peu plus que StringSet(253h). Cependant, PSSO prend moins de 32 heures. La Table 4.14 rapporte les temps d'exécution moyens par recherche de clones. Nous pouvons voir que le temps de prétraitement de PSS peut parfois être important, par exemple sur les grands programmes Windows avec 16,95s contre 2,22s dépensées pour les calculs de similarité. Tout d'abord, il faut noter que le temps de prétraitement n'augmente pas avec la taille du dépôt. Deuxièmement, PSSO est particulièrement optimisé pour de tels cas et son temps de prétraitement reste faible.

Conclusion (RQ1) PSS est souvent à peu près aussi rapide que MutantX-S sur les jeux de données importants, mais il peine sur les programmes Windows de grande taille présent . PSSO remédie à ce défaut et est systématiquement plus rapide que les autres approches, à l'exception de FunctionSet. Étonnamment, StringSet est lent sur les trois grands jeux de données.

TABLE 4.13 – (RQ1) Temps totaux sur les grands jeux de données, incluant le prétraitement. Prétraitement significatif entre parenthèse.

Jeu de données	BinKit	IoT	Windows
# de programmes	97 760	19 959	84 992
B_{size}	43h	47m	8h41m
D_{size}	43h	47m	8h45m
Shape	21h25m	21m26s	4h16m
ASCG	143h	1h23m	243h
prétraitement	(81h)	(19m12s)	(228h)
MutantX-S	220h	3h34m	41h
PSS	190h	2h9m	263h
prétraitement	(81h)	(16m42s)	(233h)
PSSO	116h	2h12m	31h29m
prétraitement	(14h3m)	(33m3s)	(5h23m)
LibDX	1965h	7h47m	170h
StringSet	542h	9h21m	253h
FunctionSet	37h	7m47s	27h34m

TABLE 4.14 – (RQ1) Temps par recherche de clones en secondes sur les grands jeux de données, incluant le prétraitement. Prétraitement significatif entre parenthèse.

Jeu de données	BinKit	IoT	Windows
# de programmes	97 760	19 959	84 992
B_{size}	0,11	0,14	0,63
D_{size}	0,11	0,14	0,63
Shape	0,05	0,06	0,31
ASCG	0,37 (0,21)	0,25 (0,06)	17,68 (16,60)
MutantX-S	0,57	0,64	3
PSS	0,49 (0,21)	0,39 (0,05)	19,17 (16,95)
PSSO	0,30 (0,04)	0,40 (0,10)	2,29 (0,39)
LibDX	5,09	1,40	12,43
StringSet	1,40	1,69	18,47
FunctionSet	0,10	0,03	2,01

4.5.4 RQ2 : Évaluation de la précision

Nous calculons des scores de précision sur les grands jeux de données. Nous présentons les résultats dans la Table 4.15.

TABLE 4.15 – (RQ2) Score de précision sur les grands jeux de données.

Jeu de données	BinKit	IoT	Windows
# de programmes	97 760	19 959	84 992
B_{size}	0,166	0,819	0,196
D_{size}	0,062	0,787	0,445
Shape	0,297	0,818	0,388
ASCG	0,554	0,759	0,444
MutantX-S	0,354	0,870	0,472
PSS	0,619	0,863	0,475
PSSO	0,619	0,862	0,466
LibDX	0,882	0,707	0,044
StringSet	0,970	0,922	0,501
FunctionSet	0,500	0,624	0,426

BinKit PSS et PSSO obtiennent un score de $0,619$ sur BinKit, tandis que l'autre méthode spectrale ASCG n'a qu'un score de $0,554$. MutantX-S est derrière avec $0,354$. En fait, nous montrons dans la Table 4.16 qu'il obtient des scores de $0,01$ en cas de changement d'architecture ainsi que contre les offuscations. Avec des identificateurs littéraux, StringSet obtient $0,970$ et LibDX $0,882$. La méthode FunctionSet a un score de seulement $0,500$.

IoT PSS a un score de $0,863$ sur IoT, proche de MutantX-S $(0,870)$. PSSO est très proche avec $0,862$, tandis que ASCG obtient $0,759$. Avec des identificateurs littéraux, StringSet obtient un score de $0,922$. Les autres méthodes utilisant des identificateurs littéraux ont des difficultés. FunctionSet a un score de $0,624$,. En effet, seulement très peu d'appels externes sont présents dans des micrologiciels. De plus, LibDX obtient $0,707$ car LibDX extrait des valeurs de chaînes constantes depuis des sections de programme en lecture seule, qui sont rares à l'intérieur des micrologiciels IoT.

Windows PSS obtient un score de $0,475$ sur Windows, juste au-dessus de *MutantX-S* $(0,472)$ et bien au-dessus de ASCG $(0,444)$. PSSO est un peu en arrière par rapport à PSS et MutantX-S avec $0,466$. Parmi les méthodes avec des identificateurs littéraux, StringSet obtient $0,501$. LibDX obtient seulement $0,044$. Encore une fois, les sections en lecture seule bien définies ne sont pas prévalentes dans les programmes Windows. Comme auparavant, FunctionSet a un score relativement faible, ne dépassant pas $0,426$.

Conclusion (RQ2) PSS et PSSO sont généralement aussi précis que MutantX-S, à l'exception des situations de changement d'architecture présentes dans le jeu de données BinKit [85] ou d'offuscations proposé par LLVM-Obfuscator [81], dans lesquelles MutantX-S échoue. Lorsque les identificateurs littéraux sont pertinents, StringSet est la méthode la plus précise dans tous les jeux de données, tandis que FunctionSet et LibDX ont des difficultés sur les jeux de données IoT et Windows.

4.5.5 RQ3 : Évaluation de la robustesse

La dernière évaluation mesure la robustesse des dix méthodes de recherche de clones qui ont survécu au test de vitesse. L'évaluation s'appuie sur le jeu de données BinKit. Nous considérons quatre situations avec des changements (i) de niveau d'optimisation, (ii) de compilateur, (iii) d'architecture et (iv) de la présence d'offuscations.

Résultats Nous écrivons les scores des champs de test les plus importants dans la Table 4.16. Lorsque des identificateurs littéraux sont disponibles, StringSet et LibDX sont très stables. FunctionSet est stable sauf dans les situations de changement d'architectures, car les noms de fonctions externes diffèrent entre les architectures. Notez qu'une limitation importante à cette constatation est que les offuscations utilisées ne cachent pas ou ne cryptent pas les chaînes de caractères littérales ou les appels externes (API). PSS et PSSO sont beaucoup plus robustes que MutantX-S dans les situations de changement d'architecture et d'offuscation. Par exemple, MutantX-S tombe à $0,02$ de l'architecture ARM à MIPS, tandis que PSS maintient un score de $0,39$. ASCG, la méthode spectrale la plus simple, tombe également à $0,08$ dans ce scénario. Étonnamment, PSS et PSSO sont meilleurs lors d'un changement d'architecture que dans les champs de test (-O0, -O3) et (-O0, -O2). Nous supposons que l'architecture n'impacte pas tant que ça le graphe d'appels en comparaison des optimisations avancées comme l'extension inline qui est activée à partir du niveau d'optimisation -O2 par les compilateurs GCC et Clang.

Conclusion (RQ3) PSS et PSSO sont robustes aux situations de changement de niveau d'optimisation, de changement de compilateur, de changement d'architecture et d'offuscation, tandis que MutantX-S subit une perte de précision importante dans les cas de changement d'architecture et d'offuscation.

4.5.6 RQ4 : Impact des composantes de PSS

Dans la Table 4.17, nous écrivons les scores de précision des deux composantes de PSS : simCG et simCFG. Pour rappel, la première composante est une distance entre les valeurs propres des graphes d'appels, tandis que la seconde est une distance entre les nombres d'arêtes des graphes de flot de contrôle des fonctions. De façon notable, PSS atteint toujours un meilleur score de précision que ces composantes sur les grands jeux de données. Nous remarquons que la composante simCFG seule n'est pas précise sur le jeu de données Windows ($0,163$ vs $0,459$ pour simCG).

TABLE 4.16 – (RQ2, RQ3) Scores de précision sur BinKit.

Catégorie	Niveau d'opt.						Chang. de compilateur			Chang. d'architecture				vs Offuscation†			
vs	O0 O1	O0 O2	O0 O3	O1 O2	O1 O3	O2 O3	gcc-4 gcc-8	clang-4 clang-7	clang gcc	ARM MIPS	ARM x86	MIPS x86	32 64	bcf	fla	sub	all
B_{size}	0,04	0,04	0,07	0,19	0,11	0,21	0,11	0,45	0,07	0,03	0,10	0,04	0,04	0,04	0,01	0,08	0,01
D_{size}	0,03	0,03	0,03	0,06	0,05	0,07	0,07	0,09	0,04	0,02	0,05	0,03	0,04	0,02	0,01	0,05	0,01
Shape	0,19	0,07	0,06	0,17	0,11	0,33	0,38	0,65	0,16	0,04	0,16	0,04	0,19	0,25	0,27	0,48	0,23
ASCG	0,40	0,12	0,10	0,43	0,24	0,68	0,78	0,91	0,46	0,08	0,46	0,06	0,59	0,54	0,64	0,78	0,48
MutantX-S	0,04	0,03	0,03	0,43	0,36	0,64	0,67	0,80	0,14	0,02	0,01	0,01	0,06	0,09	0,03	0,54	0,01
PSS	0,54	0,23	0,17	0,59	0,38	0,70	0,79	0,91	0,51	0,39	0,55	0,39	0,66	0,53	0,57	0,82	0,46
PSSO	0,53	0,24	0,17	0,60	0,39	0,68	0,78	0,90	0,51	0,44	0,54	0,44	0,66	0,52	0,56	0,82	0,46
LibDX	0,89	0,89	0,89	0,89	0,89	0,89	0,89	0,86	0,78	0,87	0,89	0,90	0,88	0,87	0,86	0,86	0,86
StringSet	0,97	0,97	0,97	0,97	0,97	0,97	0,97	0,97	0,97	0,96	0,98	0,96	0,97	0,96	0,97	0,96	0,97
FunctionSet	0,55	0,53	0,53	0,55	0,55	0,56	0,46	0,68	0,55	0,29	0,02	0,00	0,23	0,61	0,61	0,61	0,61

† : Le jeu de données BinKit ne considère pas d'obsfucation des identificateurs littéraux.

Dans la Table 4.18, nous écrivons les temps moyens des recherches de clones de chaque composante. Comme attendu, les temps de PSS sont l'addition des temps de simCG et de simCFG. Ce qui veut dire que PSS est une seconde plus lent que la composante simCG dans le pire cas.

TABLE 4.17 – (RQ4) Scores de précision des composantes sur les grands jeux de données.

Jeu de données	BinKit	IoT	Windows
simCG	0,596	0,856	0,459
simCFG	0,424	0,856	0,163
PSS	0,619	0,863	0,475

TABLE 4.18 – (RQ4) Temps par recherche de clones (sec) des composantes sur les grands jeux de données. Inclut le prétraitrement. Prétraitement significatif entre parenthèse.

Jeu de données	BinKit	IoT	Windows
simCG	0,36 (0,21)	0,22 (0,05)	18,07 (16,95)
simCFG	0,14	0,16	1,06
PSS	0,49 (0,21)	0,39 (0,05)	19,17 (16,95)

Conclusion (RQ4) PSS est plus précise que ses composantes.

4.6 Résumé de nos résultats principaux

Nos nouvelles méthodes spectrales PSS et PSSO atteignent un point optimal quant à l'équilibre entre la vitesse, la précision et la robustesse. Elles n'ont pas besoin de phase d'entraînement, s'adaptent très bien à de grands dépôts et sont très robustes, même dans des situations de changement d'architecture ou de compilateur ainsi qu'en cas d'offuscation légère. Par conséquent, elles sont les meilleures candidates pour une recherche intensive de clones de programmes.

Il est également important de mentionner que les adaptations standard des méthodes spectrales basées sur des graphes manquent de précision par rapport à PSS et que l'optimisation PSSO est nécessaire pour de grands programmes.

Un résumé des résultats est donné par la Table 4.19.

Notre étude approfondie a également permis de mettre en évidence que la plupart des approches précédentes dans le domaine [42, 101, 106, 154], principalement axées sur la similarité au niveau des fonctions, sont trop lentes pour rechercher des clones de programmes.

TABLE 4.19 – Résumé informel des résultats expérimentaux.

Méthode	Vitesse	Précision	Robustesse	Attention à
ASCG [54]	+	-	+	
MutantX-S [68]	+	+	--	
PSS/PSSO	+/++	+	+	
LibDX [140]	-	++	++	Extraction des chaînes Offuscation des chaînes
StringSet	--	+++	++	Offuscation des chaînes
FunctionSet	+++	-	-	Offuscation des appels externes Bibliothèque statique

4.7 Limitations

Bien que PSS et PSSO obtiennent de bons résultats dans nos expériences, il y a tout de même des cas particuliers à considérer.

De façon générale, ces méthodes souffrent des grandes différences entre les graphes d'appels de la cible et des clones à retrouver. Cela peut être causé par plusieurs phénomènes :

- Un changement important du code source – c'est pourquoi nous ne gérons que des modifications incrémentales d'une application ou d'une bibliothèque ;
- Des optimisations interprocédurales agressives, telles que l'extension inline ou le partage d'une fonction – ces optimisations pourraient être un problème grandissant en cas de progrès des compilateurs ;
- Des offuscations interprocédurales agressives, telles que la fusion des fonctions ou la virtualisation.

De plus, les programmes que nous avons utilisés proviennent très largement de codes sources C/C++. Il serait intéressant d'évaluer les méthodes de recherche de clones sur des programmes écrits dans des langages de programmation émergents tels que Rust ou Go. En effet, ces langages pourraient fournir d'autres façons de gérer les appels de fonctions.

4.8 Conclusion

Nous considérons le problème de la recherche de clones de programmes dans de grands dépôts de programme. Bien que de nombreux travaux aient été consacrés à la similarité des fonctions, les rares techniques existantes pour la similarité de programmes souffrent de problèmes de vitesse sur de tels grands jeux de données, ou d'une faible précision, ou encore d'une faible robustesse face aux variations des codes binaires. Nous proposons une nouvelle méthode appelée Program Spectral Similarity (PSS), et en particulier sa version optimisée PSSO, qui atteint un point optimal en termes de vitesse, de précision et de robustesse, démontrant à la fois une grande vitesse et une bonne précision même dans les situations de changement de compilateur ou d'architecture.

Chapitre 5

Étude complémentaire : recherche rapide de clones

Le Chapitre 5 est une étude préliminaire qui vise à augmenter la vitesse de détection des clones dans des dépôts de programmes de grande taille, tout en veillant à maintenir la précision et la robustesse. La stratégie principale repose sur l'utilisation d'une structure de données spécifiquement conçue pour accélérer la recherche. Cette structure permet une partition efficace de l'espace de recherche. Pour l'utiliser, il est nécessaire de modifier notre méthode spectrale afin que son prétraitement produise un vecteur de dimension fixe. Mutant-S est lui directement compatible. Cette étude ne bouleverse pas les conclusions précédentes, notre invention PSSOH a toujours l'avantage de la robustesse ainsi qu'une bonne précision tout en étant encore plus rapide que PSS et PSSO.

5.1 Procédure de recherche rapide de clones

Nous modifions les procédures de recherche de clones afin de les accélérer sur les dépôts larges. Nous appelons cela des *procédures de recherche rapide de clones*. L'idée générale suivie jusque là était d'améliorer l'efficacité du calcul de similarité. Nous voulons maintenant diminuer le nombre de calculs de similarité, en donnant au dépôt de programmes une structure qui va aider la recherche de programmes similaires.

Pour cela, nous voulons intégrer un algorithme de recherche du plus proche voisin à ces procédures, ce qui permet de ne pas avoir à calculer des indices de similarité entre la cible et chaque élément du dépôt mais seulement entre la cible et des éléments jugés suffisamment proches. En effet, l'algorithme des plus proches voisins donne une structure particulière au dépôt via des méthodes de partitionnement de l'espace.

Deux hypothèses sont nécessaires pour utiliser les procédures de recherche rapide de clones :

1. le prétraitement de la métrique de similarité extrait depuis le programme P un vecteur V de dimension fixe ;
2. la similarité entre deux programmes est reliée à la distance entre leurs vecteurs associés.

Pour obtenir une procédure rapide avec PSS, il nous faut modifier notre méthode spectrale puisque celle-ci produit deux vecteurs dont les dimensions ne sont pas fixes. Naturellement, nous sélectionnons une partie de ces vecteurs et les réunissons dans un unique vecteur de taille fixe.

Formellement Étant donné un programme inconnu *cible* P et un *dépôt* de programmes D, l'objectif est toujours d'identifier un *clone* de P dans D. Mais, cette fois, l'élément le plus spécifique à une procédure est son prétraitement. Ce prétraitement extrait nécessairement un vecteur de dimension fixe pour un programme. Le programme P est donc associé à un vecteur V. De plus, le dépôt est organisé selon une structure spécifique. En effet, une instance S d'une structure de donnée *Struct* est calculée à partir du dépôt D. L'usage de S permet d'accélérer la recherche du plus proche voisin de V. Pour cela, il existe un algorithme *Algo*, spécifique à la structure de données *Struct*, qui permet d'effectuer une recherche d'un vecteur proche de V dans S sans avoir à calculer la distance entre V et les vecteurs dans S. Cette recherche est approchée, c.a.d. que le vecteur trouvé par *Algo* n'est pas nécessaire le plus proche de V dans S.

Étapes À haut niveau, le processus de requête est divisé en trois étapes :

1. **Prétraitement de la requête.** Lors de la requête, nous recevons le programme cible P. Nous effectuons un prétraitement à cette étape, c.a.d. que nous extrayons un vecteur V de dimension fixe caractérisant P construit par rapport à l'instance S de la structure de données *Struct* ;
2. **Recherche.** En appliquant l'algorithme *Algo* sur S et le vecteur V, nous trouvons un vecteur V_m proche de V dans S ;
3. **Décision.** Le vecteur V_m est associé au programme Q_m. Ce programme Q_m est considéré comme un clone potentiel. La recherche de clones est un succès si Q_m est un clone de P, sinon c'est un échec.

La Figure 5.1 illustre l'architecture d'une procédure de recherche rapide de clones.

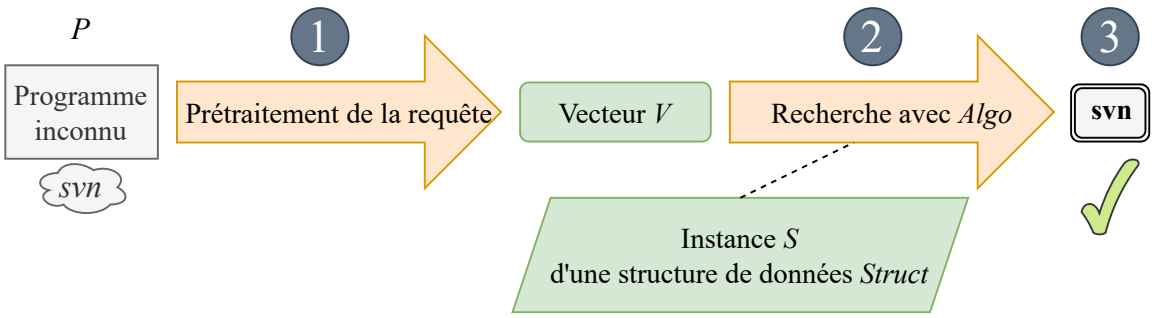


FIGURE 5.1 – Architecture de la procédure de recherche rapide de clones.

Défis Les défis précédents des recherches de clones sont modifiés par l'utilisation d'une structure de données. En principe, l'efficacité est augmentée par rapport à une recherche classique. Mais

le temps de calcul de l'instance de la structure de données dépend à la fois de la taille des programmes et de la taille du dépôt. Il ne faut pas que celui-ci soit trop coûteux ou que le gain de temps durant la recherche de clones soit trop bas. De plus, la précision est réduite par le fait que la recherche du vecteur le plus proche soit approximative. Nous cherchons naturellement à ce que cette perte soit basse. En somme, la perte de précision doit être jugée acceptable par rapport à un gain de temps durant la recherche de clones.

5.2 Structure de données et algorithme de recherche

Nous introduisons dans cette Section la structure de données et l'algorithme de recherche liée qui sont tous deux proposés par l'outil Annoy [19] .

5.2.1 Bagage scientifique

Charikar [31] a introduit une technique appelée SimHash basée sur les projections aléatoires. Son principe de base est de choisir un hyperplan aléatoire et d'utiliser cet hyperplan pour hacher les vecteurs de réels en entrée. Soit un vecteur V et un hyperplan H défini par un vecteur unitaire normalisé R, on définit $h(V) = signe(V \cdot R)$. C'est-à-dire, $h(V) = \pm 1$ selon que V se trouve d'un côté ou de l'autre de l'hyperplan. Chaque hyperplan aléatoire peut alors produire un hachage différent. Pour deux vecteurs u, v d'angle θ, il est prouvé [31] que $P(h(u) = h(v)) = 1 - \frac{\theta}{\pi}$. Ainsi la probabilité que les hachés de deux vecteurs soit les mêmes est proportionnelle à leur similarité angulaire.

5.2.2 Annoy

L'outil Annoy est construit d'après les idées de Charikar. Il fournit pour structure de données une forêt d'arbres binaires ainsi qu'un l'algorithme spécifique à la structure afin de faire des recherches rapides en explorant la forêt de façon parcimonieuse. En contrepartie de la rapidité des recherches de clones, il ne fournit aucune garantie théorique sur proximité des vecteurs retrouvés. Néanmoins, en pratique, il est efficace. Annoy est utilisé par Spotify pour trouver des musiques proches les unes des autres.

Fonctionnement global de l'algorithme L'algorithme commence par construire A arbres binaires dans lesquels un nœud partitionne les vecteurs dans le dépôt en deux groupes par rapport à un plan aléatoire. Les feuilles contiennent un nombre maximum de L vecteurs (p. ex., 100). Pendant la recherche, tous les arbres sont explorés avec une file de priorité. Les nœuds proposant la meilleure séparation (la plus grande distance entre la cible et l'hyperplan associé aux nœuds) sont explorés en premier. Lorsqu'une feuille est rencontrée, l'algorithme collecte les vecteurs associés à la feuille. L'exploration s'arrête quand K vecteurs ont été collectés. Ensuite, une recherche exhaustive du plus proche de la cible est faite parmi les K vecteurs collectés.

19. `https://github.com/spotify/annoy`

Création d'un arbre La racine de l'arbre référence tous les vecteurs. Tant qu'un nœud référence plus de L vecteurs, deux vecteurs du nœud sont pris au hasard, l'hyperplan équidistant entre les deux est calculé et associé au nœud. Deux nœuds enfants sont ensuite ajoutés, un référençant les vecteurs à gauche de l'hyperplan et un à droite. Les nœuds sans enfants sont les feuilles de l'arbre.

Recherche d'un vecteur proche de U On initialise la file de priorité F avec toutes les racines des arbres à priorité maximum. On initialise également S, l'ensemble des vecteurs collectés. Tant que F n'est pas vide et que S a moins de K vecteurs, le nœud X avec la plus haute priorité est sorti de la file. Si X est une feuille, les vecteurs de X sont ajoutés à S, sinon, on calcul si U est à gauche ou à droite de l'hyperplan de X, on ajoute à la file l'enfant correspondant avec pour priorité la distance entre U et l'hyperplan plan de X.

Complexité en temps Soit A le nombre d'arbres, K le nombre de vecteurs collectés durant la phase de recherche et $|D|$ le nombre de programmes dans le dépôt D. La création des arbres prend un temps $O(A \times |D| \times log_2(|D|))$. La recherche du plus proche voisin prend un temps $O(A \times K \times log_2(|D|))$. Le temps de création des arbres est donc garanti comme inférieur au calcul de toutes les distances entre les vecteurs dont le temps est $O(|D|^2)$. De plus, la recherche est rapide puisqu'elle a un temps logarithmique par rapport à la taille du dépôt.

5.3 Méthodologie

Nous souhaitons créer des méthodes rapides qui utilisent l'outil Annoy. Nous pouvons soit trouver des méthodes normales dont le prétraitement produit déjà un vecteur de dimension fixe, soit adapter les prétraitements des méthodes normales.

5.3.1 PSSOH

Notre proposition PSS produit deux vecteurs et la métrique de similarité associée consiste en des distances euclidiennes sur deux vecteurs de dimensions variables (voir la Section 3.7.2). Pour pouvoir appliquer l'outil Annoy, nous devons transformer le prétraitement de PSS afin de n'avoir qu'un seul vecteur de taille fixe. Il est intuitif de ne calculer que les 100 plus grandes valeurs propres du graphe d'appels de fonctions, comme le fait déjà PSSO. Mais PSSO prend également en compte un deuxième vecteur de dimension non borné, celui qui contient les nombres d'arêtes normalisés des CFG des fonctions. Nous résolvons le problème en introduisant PSSOH, un prétraitement qui produit un vecteur de dimension 200. Les 100 premières valeurs du vecteur de PSSOH sont celles du premier vecteur de PSSO. Le reste du vecteur de PSSOH contient les 100 plus grands nombres d'arêtes des CFG normalisés. En fait, nous concaténons les deux vecteurs de PSS après avoir réduit leur dimension à 100 en ne retenant que leurs plus grandes valeurs. PSSOH utilise l'implémentation officielle d'Annoy. Le nombre maximum d'éléments lié à une feuille d'un arbre L est automatiquement réglé suivant les capacités mémoires du processeur.

Après de premiers essais, nous assignons le nombre d'arbres A à 40 et le nombre de vecteurs collectés durant la phase de recherche K à 40 également.

5.3.2 MutantXSH

Le prétraitement de MutantX-S [68] produit un vecteur de dimension fixe dont les valeurs sont les hachés des occurrences des opcodes dans le programme (voir la Section 4.2). Le prétraitement de MutantX-S est directement utilisable avec l'outil Annoys. Nous créons MutantXSH, la version rapide de MutantX-S avec le même paramétrage d'Annoy que pour PSSOH.

5.4 Expériences

Nous effectuons des recherches rapides de clones avec nos quatre nouvelles méthodes. Pour cela, nous utilisons nos quatre jeux de données du Chapitre précédent (voir les Sections 4.4.1 et 4.5.1). Nous comparons les résultats des nouvelles méthodes avec leurs équivalents sans approximation en termes de vitesse, de précision et de robustesse.

5.4.1 Vitesse

Dans la Table 5.1, nous rapportons les temps pour une recherche de clones suivant les méthodes et les jeux de données. Une recherche avec PSSOH prend moins d'une demi-seconde sur tous les jeux de données alors qu'elle prend $2,29$s avec PSSO sur le jeu de données Windows. En fait, PSSOH ne conserve que les temps de prétraitement de PSSO et fait disparaitre tout autre temps nécessaire. MutantXSH qui n'a pas de prétraitement a un temps de recherche de $0,01$s sur tous les jeux de données alors que MutantX-S prenait 3s sur le jeu de données Windows.

Table 5.1 – Temps par recherche de clones en secondes incluant des recherches approximées, incluant le prétraitemeny de la cible. Prétraitement significatif entre parenthèse.

Jeu de données	Basique	BinKit	IoT	Windows
Nombre de programmes	950	97 760	19 959	84 992
MutantX-S	$< 0,01$	0,57	0,64	3
MutantXSH	$< 0,01$	0,01	0,01	0,01
PSS	1,41 (1,41)	0,49 (0,21)	0,39 (0,05)	19,17 (16,95)
PSSO	0,27 (0,27)	0,30 (0,04)	0,40 (0,10)	2,29 (0,39)
PSSOH	0,27 (0,27)	0,04 (0,04)	0,10 (0,10)	0,39 (0,39)

Conclusion PSSOH et MutantXSH sont effectivement très rapides puisque la taille du dépôt de programme n'a quasiment plus d'impact sur le temps de recherche. En effet, la recherche prend un temps logarithmique par rapport à la taille du dépôt.

5.4.2 Précision

Dans la Table 5.2, nous rapportons les scores de précision suivant les méthodes et les jeux de données. Entre PSS et PSSOH, il y a une baisse du score de précision de $0,01$ sur Basique ; $0,038$ sur BinKit ; $0,002$ sur IoT ; et $0,019$ sur Windows. Nous observons que MutantXSH obtient lui à peu près le même score que MutantX-S. Si on compare PSSOH et MutantXSH, on remarque que PSSOH est bien plus précise sur BinKiT ($0,583$ vs $0,353$) tandis que MutantX-S est un peu plus précise sur Windows ($0,472$ vs $0,456$) et sur IoT ($0,866$ vs $0,861$).

TABLE 5.2 – Scores de précision avec des recherches approximées.

Jeu de données	Basique	BinKit	IoT	Windows
# de Programmes	950	97 760	19 959	84 992
MutantX-S	0,38	0,354	0,870	0,472
MutantXSH	0,38	0,353	0,866	0,472
PSS	0,38	0,619	0,863	0,475
PSSO	0,38	0,619	0,862	0,466
PSSOH	0,37	0.583	0,861	0,456

Conclusion Une perte de précision des méthodes rapides est présente. Cette perte est plus visible (bien que toujours légère) pour notre méthode spectrale PSSOH que pour MutantXSH. Néanmoins, de par ses bons résultats sur le jeu de données BinKit, PSSOH reste plus précis que MutantXSH.

5.4.3 Robustesse

Nous considérons quatre situations avec des changements de niveau d'optimisation, de compilateur, d'architecture et la présence d'offuscations grâce au jeu de données BinKit. Nous écrivons les scores de certains champs de test dans les Tables 5.3 et 5.4. Nous remarquons que PSSOH est un peu moins robuste que PSS, par exemple le score de précision de PSSOH n'est plus que de $0,46$ face à l'offuscation du bug de flot de contrôle au lieu de $0,53$ avec PSS. Cependant, PSSOH conserve globalement une très bonne robustesse. Les résultats de MutantXSH et de MutantX-S sont eux quasiment identiques, et toujours assez bas. Par exemple, sur le champ de test évoqué précédemment, MutantX-S a un score de $0,09$ et MutantXSH de $0,08$. La robustesse de MutantXSH est aussi faible que celle de MutantX-S ; cette méthode ne résiste pas aux changements d'architectures et aux offuscations.

Conclusion La robustesse de PSSOH est élevée bien qu'un peu plus basse que celle de PSS. Néanmoins, PSSOH reste bien plus robuste que MutantX-S et MutantXSH.

TABLE 5.3 – Scores de précision sur BinKit avec des recherches approximées - Partie 1.

Catégorie	Niveau d'optimisation						Changement de compilateur		
vs	O0 O1	O0 O2	O0 O3	O1 O2	O1 O3	O2 O3	GCC-4 GCC-8	Clang-4 Clang-7	Clang GCC
MutantX-S	0,04	0,03	0,03	0,43	0,36	0,64	0,67	0,80	0,14
MutantXSH	0,04	0,03	0,03	0,44	0,36	0,64	0,67	0,80	0,14
PSS	0,54	0,23	0,17	0,59	0,38	0,70	0,79	0,91	0,51
PSSO	0,53	0,24	0,17	0,60	0,39	0,68	0,78	0,90	0,51
PSSOH	0,49	0,22	0,16	0,58	0,35	0,61	0,75	0,88	0,45

TABLE 5.4 – Scores de précision sur BinKit avec des recherches approximées - Partie 2.

Catégorie	Changement d'architecture				vs Offuscation			
vs	ARM MIPS	ARM x86	MIPS x86	32 64	bcf	fla	sub	all
MutantX-S	0,02	0,01	0,01	0,06	0,09	0,03	0,54	0,01
MutantXSH	0,02	0,01	0,02	0,06	0,08	0,03	0,54	0,01
PSS	0,39	0,55	0,39	0,66	0,53	0,57	0,82	0,46
PSSO	0,44	0,54	0,44	0,66	0,52	0,56	0,82	0,46
PSSOH	0,38	0,51	0,37	0,62	0,46	0,47	0,80	0,40

5.5 Conclusion

Cette étude préliminaire des méthodes rapides ne modifie pas les grandes tendances évoquées en conclusions du Chapitre précédent. Dans la Table 5.5, nous résumons les points forts et faibles des différentes méthodes en incluant les méthodes rapides.

TABLE 5.5 – Résumé informel des résultats expérimentaux incluant les recherches rapides.

Méthode	Vitesse	Précision	Robustesse
PSS/PSSO	+/++	+	+ +
PSSOH	+ + +	+	+ +
MutantX-S	+	+	- -
MutantXSH	+ + + +	+	- -

Notre méthode spectrale PSSOH est encore plus rapide, et ne perd que très légèrement en précision et en robustesse. La méthode MutantXSH devient la plus rapide, mais reste très peu robuste, ce qui rend cette méthode moins intéressant que PSSOH qui combine vitesse, précision et robustesse.

Conclusion et perspectives

L'accès au code source d'un programme n'est pas toujours possible. Par exemple, dans le cas des programmes malveillants, ou des logiciels dont les entreprises cherchent à protéger la propriété intellectuelle, ou des programmes encore utilisés dont le code source a été perdu avec le temps. Face à ces défis, la rétro-ingénierie des programmes est nécessaire.

Au cours de cette thèse, nous avons vu comment prolonger la rétro-ingénierie à bas niveau par des analyses automatiques de programmes plus haut niveau afin de s'informer sur un programme entier. Nous pensons que ces analyses peuvent contribuer à une meilleure compréhension des programmes. Ces analyses constituent une ligne de défense essentielle contre les programmes malveillants, ainsi qu'une protection potentielle de la propriété intellectuelle.

Nous mettons en évidence la nécessité d'une vue globale du code binaire là où les approches actuelles focalisées sur l'échelle des fonctions souffrent de plusieurs limites. D'une part, les programmes peuvent contenir des milliers de fonctions distinctes ayant des interrelations complexes dont la prise en compte est nécessaire à l'analyse. D'autre part, compte tenu de l'abondance de programmes présents dans l'écosystème informatique, l'analyse doit être rapide, ce qui n'est pas aisé si on procède en analysant chaque fonction séparément.

1 Résumé de nos contributions

Nous nous attaquons à deux tâches de recherches d'informations : la prédiction de la chaîne de compilation et la recherche d'un clone de programme. Nous apportons de nouvelles solutions.

- Premièrement, nous proposons Site Neural Network (SNN), une approche basée sur les réseaux de neurones sur les graphes (GNN), pour prédire la chaîne de compilation utilisée pour générer un code binaire donné. L'originalité de cette approche repose sur le fait qu'elle cible les programmes dans leur globalité, sans se limiter aux instructions ou aux prologues de fonctions. Nous mettons en place un cadre d'évaluation comprenant plus de 36 000 programmes provenant de 23 versions différentes de quatre compilateurs (Clang, GCC, MinGW et Visual Studio), et avec jusqu'à quatre niveaux d'optimisations. Par rapport à une autre méthode de l'état de l'art, SNN offre de meilleurs résultats en termes de précision et de temps d'apprentissage, tout en permettant l'intégration de nouvelles chaînes de compilation de manière efficace via des hiérarchies de classifieurs ;

- Deuxièmement, nous proposons la méthode Program Spectral Similarity (PSS) pour la recherche de clones de programmes sur de larges dépôts. Le point clé de cette méthode est de créer une représentation d'un programme à partir de l'analyse spectrale de son graphe d'appels de fonctions. Nous mettons en place un cadre d'évaluation de 15 méthodes de recherches de clones avec plus de 200 000 programmes, comprenant des programmes Linux et Windows ainsi que des programmes malveillants ciblant les objets connectés. Notre approche est à la fois rapide, précise et robuste, offrant ainsi une bonne mesure de similarité de programmes. De plus, nous proposons deux variantes PSSO et PSSOH afin d'accélérer encore la méthode si nécessaire contre une perte faible de précision. Enfin, nous avons au passage démontré la pertinence de différencier les méthodes nécessitant des identificateurs littéraux (tels que les chaînes littérales) et celles n'en requérant pas. Dans le premier cas, nous suggérons deux méthodes : FunctionSet et StringSet, cette dernière surpassant l'état de l'art en matière de précision.

Dans l'ensemble, cette thèse montre que l'analyse des programmes complets peut être effectuée en temps court avec précision et robustesse par la mise en œuvre de nos techniques innovantes SNN et PSS. Par ailleurs, elle met en évidence l'existence de lacunes dans les méthodes existantes lors de la recherche de clones de programmes. Nos travaux contribuent au domaine de l'analyse automatisée des programmes, afin de faire face à la variété et la complexité croissantes des systèmes informatiques. Cette thèse ouvre deux voies principales : la vectorisation des programmes avec de l'apprentissage, et l'utilisation des spectres de graphes pour l'étude des similarités.

2 Perspectives

Nous présentons maintenant quelques perspectives de recherches basées sur nos travaux. Nous commençons par les perspectives d'apprentissage automatique, et continuons sur les perspectives d'étude des spectres de graphes. Enfin, nous présentons des applications directes de nos travaux.

Apprentissage de vecteurs de programmes Dans le Chapitre 2, nous apprenons un modèle de vectorisation de programmes dans le but de détecter les chaînes de compilation. Il serait intéressant de considérer différemment la vectorisation de programmes entiers :

- Créer de meilleurs réseaux de neurones sur les graphes est un problème de recherche actuel. Nos réseaux SNN utilisent surtout des idées venues du champ de la reconnaissance d'image tels que les résidus [162], ou des couches de regroupements par le maximum [135]. Il est nécessaire d'inventer des méthodes spécifiques aux graphes provenant de programmes ;
- Les vecteurs de programmes de SNN ne sont qu'un intermédiaire afin de déterminer la chaîne de compilation du programme. Nous pourrions ajouter d'autres tâches inspirées du langage naturel comme la prédiction de la suite du programme d'après un fragment à la manière de jTrans [148]. En utilisant une analyse dynamique, on pourrait créer des tâches demandant de prédire le comportement du programme.

Analyse spectrale Dans le Chapitre 3, nous présentons PSS, notre méthode spectrale qui compare structurellement les graphes d'appels et les graphes de flot de contrôle des fonctions. On peut étendre PSS de plusieurs façons :

- Actuellement PSS ne considère les CFG qu'à travers les nombres d'arêtes qu'ils contiennent. Il est séduisant d'ajouter plus d'informations venant de ces graphes, voire de les intégrer à l'analyse spectrale. Pour cela, des outils mathématiques existent déjà [51, 66] ;
- Notre analyse spectrale ne nous permet pas actuellement de rechercher une partie d'un programme à l'intérieur d'un autre. Pourtant l'analyse spectrale est en train d'être considérée pour la recherche de sous graphes [65] ;
- Puisque nous n'avons pas utilisé d'apprentissage pour PSS, il est naturel d'y songer. Cela pourrait se faire sous la forme d'un réseau jumeau [24] qui à partir de deux spectres essaye de prédire si deux programmes sont des clones.

Applications Nous avons déjà mentionné plusieurs applications, cependant il serait intéressant d'utiliser nos approches pour de nouvelles tâches.

- On pourrait utiliser l'information de la chaîne de compilation obtenue par nos SNN pour rechercher des fonctions binaires. L'idée n'est pas tout à fait nouvelle puisqu'elle a été tentée et réussie par Vestige [79] peu après la sortie de notre article décrivant SNN ;
- De même, on pourrait utiliser la recherche de clones de programmes de PSS pour rechercher des clones de fonctions. Si on connaît le programme d'où vient la fonction cible, on peut retrouver un clone de celui-ci. Il ne reste plus qu'à trouver un clone de la fonction cible par le programme clone, ce qui est plus facile que de chercher dans le dépôt de fonctions complet ;
- PSSOH, la version de PSS avec des recherches rapides, peut être utilisée à très large échelle. On pourrait s'en servir pour détecter des programmes malveillants face à des dépôts contenant des centaines de milliers de programmes.

Bibliographie

[1] Rabe Abdalkareem, Emad Shihab, and Juergen Rilling. On code reuse from stackoverflow : An exploratory study on android apps. *Information and Software Technology*, 88 :148–158, 2017.

[2] Hadj Ahmed Bay Ahmed, Abdel-Ouahab Boudraa, and Delphine Dare-Emzivat. A joint spectral similarity measure for graphs classification. *Pattern Recognition Letters*, 2019.

[3] Jim Alves-Foss and Jia Song. Function boundary detection in stripped binaries. In *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC '19, page 84–96, New York, NY, USA, 2019. Association for Computing Machinery.

[4] Shun-ichi Amari. Backpropagation and stochastic gradient descent method. *Neurocomputing*, 5(4-5) :185–196, 1993.

[5] Blake Anderson, Daniel Quist, Joshua Neil, Curtis Storlie, and Terran Lane. Graph-based malware detection using dynamic analysis. *Journal in Computer Virology*, 2011.

[6] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA, 2016. Association for Computing Machinery.

[7] Jinrong Bai, Qibin Shi, and Shiguang Mu. A malware and variant detection method using function call graph isomorphism. *Security and Communication Networks*, 2019.

[8] Yunsheng Bai, Hao Ding, Song Bian, Ting Chen, Yizhou Sun, and Wei Wang. Simgnn : A neural network approach to fast graph similarity computation. In *Proceedings of the 12th ACM International Conference on Web Search and Data Mining*, 2019.

[9] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[10] G. Balakrishnan, Thomas Reps, David Melski, and T. Teitelbaum. *WYSINWYX : What you see is not what you execute*, volume 32, pages 202–213. 06 2008.

[11] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 2016.

[12] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. BYTEWEIGHT : Learning to recognize functions in binary code. In *23rd USENIX Security Symposium*, pages 845–860, 2014.

[13] Sébastien Bardin, Philippe Herrmann, and Franck Védrine. Refinement-based cfg reconstruction from unstructured programs. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'11, page 54–69, Berlin, Heidelberg, 2011. Springer-Verlag.

[14] Horace B Barlow. Unsupervised learning. *Neural computation*, 1(3) :295–311, 1989.

[15] Ira Baxter, Andrew Yahin, Leonardo de Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. volume 368-377, pages 368–377, 01 1998.

[16] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, page 46. Califor-nia, USA, 2005.

[17] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 2007.

[18] Tristan Benoit. Artifacts - Scalable Program Clone Search through Spectral Analysis, 2023. Available at `https://doi.org/10.5281/zenodo.8289599`.

[19] Tristan Benoit, Jean-Yves Marion, and Sébastien Bardin. Scalable program clone search through spectral analysis. In *Proceedings of the 31th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023.

[20] Tristan Benoit, Jean-Yves Marion, and Sébastien Bardin. Binary level toolchain provenance identification with graph neural networks. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 131–141, 2021.

[21] Leyla Bilge and Tudor Dumitraş. Before we knew it : An empirical study of zero-day attacks in the real world. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 833–844, New York, NY, USA, 2012. Association for Computing Machinery.

[22] Guillaume Bonfante, Jose Fernandez, Jean-Yves Marion, Benjamin Rouxel, Fabrice Sabatier, and Aurélien Thierry. CoDisasm : Medium Scale Concatic Disassembly of Self-Modifying Binaries with Overlapping Instructions. In *22nd ACM Conference on Computer and Communications Security*, Denver, United States, October 2015.

[23] Martial Bourquin, Andy King, and Edward Robbins. Binslayer : Accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, 2013.

[24] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. Signature verification using a "siamese" time delay neural network. In *Proceedings of the 6th International Conference on Neural Information Processing Systems*, 1993.

[25] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Detecting self-mutating malware using control-flow graph matching. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2006.

[26] Joan Calvet, José M. Fernandez, and Jean-Yves Marion. Aligot : Cryptographic function identification in obfuscated binary programs. In *Proceedings of the 2012 ACM CCS*, page 169–182, 2012.

[27] Donald T Campbell. Factors relevant to the validity of experiments in social settings. *Psychological bulletin*, 54(4) :297, 1957.

[28] Sylvain Cecchetto. *Analyse du flot de données pour la construction du graphe de flot de contrôle des codes obfusqués*. Theses, Université de Lorraine, February 2021.

[29] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. Bingo : Cross-architecture cross-os binary search. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016.

[30] Olivier Chapelle, Bernhard Scholkopf, and Alexander Zien. Semi-supervised learning (chapelle, o. et al., eds. ; 2006)[book reviews]. *IEEE Transactions on Neural Networks*, 20(3) :542–542, 2009.

[31] Moses S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*, STOC '02, page 380–388, New York, NY, USA, 2002. Association for Computing Machinery.

[32] Yu Chen, Zhiqiang Shi, Hong Li, Weiwei Zhao, Yiliang Liu, and Yuansong Qiao. Himalia : Recovering compiler optimization levels from binaries by deep learning. In *Intelligent Systems and Applications*, pages 35–47. Springer Publishing, 2019.

[33] Binlin Cheng, Jiang Ming, Erika A Leal, Haotian Zhang, Jianming Fu, Guojun Peng, and Jean-Yves Marion. Obfuscation-Resilient executable payload extraction from packed malware. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[34] Fan Chung. *Spectral graph theory*. American Mathematical Society, 1997.

[35] Brian Crawford, Ralucca Gera, Jeffrey House, Thomas Knuth, and Ryan Miller. Graph structure similarity using spectral graph theory. *International Workshop on Complex Networks and their Applications*, 2016.

[36] L. Daniel, S. Bardin, and T. Rezk. Binsec/rel : Efficient relational symbolic execution for constant-time at binary-level. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1021–1038, Los Alamitos, CA, USA, may 2020. IEEE Computer Society.

[37] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. BINSEC/SE : A dynamic symbolic execution toolkit for binary-level analysis. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*, pages 653–656. IEEE Computer Society, 2016.

[38] Yaniv David, Nimrod Partush, and Eran Yahav. Similarity of binaries through re-optimization. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming*

Language Design and Implementation, PLDI 2017, page 79–94, New York, NY, USA, 2017. Association for Computing Machinery.

[39] Yaniv David, Nimrod Partush, and Eran Yahav. Firmup : Precise static detection of common vulnerabilities in firmware. In *Proceedings of the 23th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018.

[40] Yaniv David and Eran Yahav. Tracelet-based code search in executables. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[41] Prasad Deshpande and Mark Stamp. Metamorphic malware detection using function call graph analysis. *MIS Review*, 2016.

[42] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. Asm2vec : Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. *IEEE Symposium on Security and Privacy (SP)*, 2019.

[43] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. Identifying open-source license violation and 1-day security risk at large scale. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[44] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. Learning program-wide code representations for binary diffing. In *27th Network and Distributed System Security Symposium*, 2020.

[45] Thomas Dullien. Structural comparison of executable objects. *Workshop on Detection of Intrusions and Malware & Vulnerability Assessment*, 2004.

[46] Thomas Dullien and Rolf Rolles. Graph-based comparison of executable objects. *SSTIC*, 2005.

[47] Chris Eagle and Kara Nance. *The Ghidra Book : The Definitive Guide*. no starch press, 2020.

[48] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.*, 44(2), mar 2008.

[49] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. Blanket execution : Dynamic similarity testing for program binaries and components. In *23rd USENIX Security Symposium*, pages 303–317, 2014.

[50] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. discovre : Efficient cross-architecture identification of bugs in binary code. In *NDSS*, 2016.

[51] Soheil Feizi, Gerald Quon, Mariana Recamonde-Mendoza, Muriel Médard, Manolis Kellis, and Ali Jadbabaie. Spectral alignment of graphs. *IEEE Transactions on Network Science and Engineering*, 7(3) :1182–1197, 2020.

[52] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[53] Aasa Feragen, Niklas Kasenburg, Jens Petersen, Marleen de Bruijne, and Karsten Borgwardt. Scalable kernels for graphs with continuous attributes. *26th International Conference on Neural Information Processing Systems*, 2013.

[54] Marc Fyrbiak, Sebastian Wallat, Sascha Reinhard, Nicolai Bissantz, and Christof Paar. Graph similarity and its applications to hardware security. *IEEE Transactions on Computers*, 2020.

[55] Debin Gao, Michael K. Reiter, and Dawn Song. Binhunt : Automatically finding semantic differences in binary programs. In *Proceedings on International Conference on Information and Communications Security*, 2008.

[56] Xinbo Gao, Bing Xiao, Dacheng Tao, and Xuelong Li. A survey of graph edit distance. *Pattern Analysis and Applications*, 2010.

[57] Ilfak Guilfanov. Ida fast library identification and recognition technology (flirt technology) : In-depth, 2012.

[58] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning*, 2020.

[59] Frank J Hall. The adjacency matrix, standard laplacian, and normalized laplacian, and some eigenvalue interlacing results. *Department of Mathematics and Statistics, Georgia State University, Atlanta*, 2010.

[60] William L Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs : Methods and applications. *arXiv preprint arXiv :1709.05584*, 2017.

[61] Irfan Ul Haq and Juan Caballero. A survey of binary code similarity. *ACM Computing Surveys (CSUR)*, 2021.

[62] Trevor Hastie, Robert Tibshirani, Jerome Friedman, Trevor Hastie, Robert Tibshirani, and Jerome Friedman. Overview of supervised learning. *The elements of statistical learning : Data mining, inference, and prediction*, pages 9–41, 2009.

[63] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[64] Armijn Hemel, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstra. Finding software license violations through binary code clone detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, 2011.

[65] Judith Hermanns, Amit Boyarski, Petros Petsinis, Alex M. Bronstein, Davide Mottin, and Panagiotis Karras. Spectral subgraph localization, 2023.

[66] Judith Hermanns, Anton Tsitsulin, Marina Munkhoeva, Alex Bronstein, Davide Mottin, and Panagiotis Karras. Grasp : Graph alignment through spectral signatures. In Leong Hou U, Marc Spaniol, Yasushi Sakurai, and Junying Chen, editors, *Web and Big Data*, pages 44–52, Cham, 2021. Springer International Publishing.

[67] Michael J. Hohnka, Jodi A. Miller, Kenrick M. Dacumos, Timothy J. Fritton, Julia D. Erdley, and Lyle N. Long. Evaluation of compiler-induced vulnerabilities. *Journal of Aerospace Information Systems*, 16(10) :409–426, 2019.

[68] Xin Hu, Sandeep Bhatkar, Kent Griffin, and Kang G. Shin. Mutantx-s : Scalable malware clustering based on static features. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, 2013.

[69] Xin Hu, Tzi cker Chiueh, and Kang G. Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2009.

[70] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. Binary code clone detection across architectures and compiling configurations. In *IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, 2017.

[71] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[72] Jianjun Huang, Bo Xue, Jiasheng Jiang, Wei You, Bin Liang, Jingzheng Wu, and Yanjun Wu. Scalably detecting third-party android libraries with two-stage bloom filtering. *IEEE Transactions on Software Engineering*, pages 1–14, 2022.

[73] Vector 35 Inc. Binary ninja. In *GitHub*. 2016.

[74] Intel. Intel 64 and ia-32 architectures software developer's manual combined volumes 2a, 2b, and 2c : Instruction set reference.

[75] Sergey Ioffe and Christian Szegedy. Batch normalization : Accelerating deep network training by reducing internal covariate shift, 2015.

[76] Emily R Jacobson, Nathan Rosenblum, and Barton P Miller. Labeling library functions in stripped binaries. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, pages 1–8, 2011.

[77] Jiyong Jang, Abeer Agrawal, and David Brumley. Redebug : Finding unpatched code clones in entire os distributions. In *2012 IEEE Symposium on Security and Privacy*, 2012.

[78] Jiyong Jang, Maverick Woo, and David Brumley. Towards automatic software lineage inference. In *Proceedings of the 22nd USENIX Conference on Security*, 2013.

[79] Yuede Ji, Lei Cui, and H. Howie Huang. Vestige : Identifying binary code provenance for vulnerability detection. In *Applied Cryptography and Network Security : 19th International Conference, ACNS 2021, Kamakura, Japan, June 21–24, 2021, Proceedings, Part II*, page 287–310, Berlin, Heidelberg, 2021. Springer-Verlag.

[80] Irena Jovanović and Zoran Stanić. Spectral distances of graphs. *Linear Algebra and its Applications*, 2012.

[81] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM – software protection for the masses. In *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15*, 2015.

[82] Boojoong Kang, Taekeun Kim, Heejun Kwon, Yangseo Choi, and Eul Gyu Im. Malware classification method via binary content comparison. In *Proceedings of the 2012 ACM Research in Applied Computation Symposium*, 2012.

[83] Paul J Kelly. A congruence theorem for trees. 1957.

[84] Wei Ming Khoo, Alan Mycroft, and Ross Anderson. Rendezvous : A search engine for binary code. In *10th Working Conference on Mining Software Repositories (MSR)*, 2013.

[85] Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel Son, and Yongdae Kim. Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. *IEEE Transactions on Software Engineering*, pages 1–23, 2022.

[86] Johannes Kinder and Dmitry Kravchenko. Alternating control flow reconstruction. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 267–282, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[87] Johannes Kinder, Florian Zuleger, and Helmut Veith. An abstract interpretation-based framework for control flow reconstruction from binaries. In Neil D. Jones and Markus Müller-Olm, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 214–228, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[88] Johannes Köbler, Uwe Schöning, and Jacobo Torán. *The Graph Isomorphism Problem : Its Structural Complexity*. Birkhauser Verlag, CHE, 1994.

[89] Orestis Kostakis, Joris Kinable, Hamed Mahmoudi, and Kimmo Mustonen. Improved call graph comparison using simulated annealing. In *Proceedings of the ACM Symposium on Applied Computing*, 2011.

[90] Nils M. Kriege, Pierre-Louis Giscard, and Richard C. Wilson. On valid optimal assignment kernels and applications to graph classification. In *30th International Conference on Neural Information Processing Systems*, 2016.

[91] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Polymorphic worm detection using structural information of executables. *International Workshop on Recent Advances in Intrusion Detection*, 2006.

[92] Harold William Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 1955.

[93] Solomon Kullback. *Information theory and statistics*. Courier Corporation, 1997.

[94] Arun Lakhotia, Mila Dalla Preda, and Roberto Giacobazzi. Fast location of similar code fragments using semantic 'juice'. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, 2013.

[95] Serge Lang. Linear algebra, 1987.

[96] Junhyun Lee, Inyeop Lee, and Jaewoo Kang. Self-attention graph pooling, 04 2019.

[97] Yeo Reum Lee, BooJoong Kang, and Eul Gyu Im. Function matching-based binary-level software similarity calculation. In *Proceedings of the 2013 Research in Adaptive and Convergent Systems*, 2013.

[98] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix : Program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 627–637, New York, NY, USA, 2017. Association for Computing Machinery.

[99] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. Graph matching networks for learning the similarity of graph structured objects. In *36th International conference on machine learning*, 2019.

[100] Martina Lindorfer, Alessandro Di Federico, Federico Maggi, Paolo Milani Comparetti, and Stefano Zanero. Lines of malicious code : Insights into the malicious software industry. In *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012.

[101] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. αdiff : Cross-version binary code similarity detection with dnn. In *33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018.

[102] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin : building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6) :190–200, 2005.

[103] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. *IEEE Transactions on Software Engineering*, 2017.

[104] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, and Yanick Fratantonio. How machine learning is solving the binary function similarity problem. In *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[105] Luca Massarelli, Giuseppe Luna, Fabio Petroni, and Leonardo Querzoni. Investigating graph embedding neural networks with unsupervised features extraction for binary analysis. *Workshop on Binary Analysis Research*, 2019.

[106] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. Function representations for binary similarity. *IEEE Transactions on Dependable and Secure Computing*, 2021.

[107] Alessio Micheli. Neural network for graphs : A contextual constructive approach. *Neural Networks, IEEE Transactions on*, 20 :498 – 511, 2009.

[108] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. Binsim : Trace-based semantic binary diffing via system call sliced segment equivalence checking. In *USENIX Security Symposium*, 2017.

[109] Jiang Ming, Dongpeng Xu, and Dinghao Wu. Memoized semantics-based binary diffing with application to malware lineage inference. In *IFIP Advances in Information and Communication Technology*, 2015.

[110] E. Moretti, G. Chanteperdrix, and A. Osorio. New algorithms for control-flow graph structuring. In *Proceedings Fifth European Conference on Software Maintenance and Reengineering*, pages 184–187, 2001.

[111] Beng Heng Ng and Atul Prakash. Expose : Discovering potential binary code re-use. In *37th IEEE Annual Computer Software and Applications Conference*, 2013.

[112] Giannis Nikolentzos, Polykarpos Meladianos, Stratis Limnios, and Michalis Vazirgiannis. A degeneracy framework for graph similarity. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence, IJCAI-18*, 2018.

[113] Irving Ojalvo and Miya Newman. Vibration modes of large structures by an automatic matrix-reduction method. *Aiaa Journal - AIAA J*, 1970.

[114] Mathilde Ollivier, Sébastien Bardin, Richard Bonichon, and Jean-Yves Marion. How to kill symbolic deobfuscation for free (or : Unleashing the potential of path-oriented protections). In *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019.

[115] Alessandro Orso. Monitoring, analysis, and testing of deployed software. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, page 263–268, New York, NY, USA, 2010. Association for Computing Machinery.

[116] Yuhei Otsubo, Akira Otsuka, Mamoru Mimura, Takeshi Sakaki, and Hiroshi Ukegawa. o-glassesx : Compiler provenance recovery with attention mechanism from a short code fragment. 01 2020.

[117] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. Sok : All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.

[118] Karl Pearson. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin philosophical magazine and journal of science*, 2(11) :559–572, 1901.

[119] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In *IEEE Symposium on Security and Privacy*, 2015.

[120] Jannik Pewny, Felix Schuster, Lukas Bernhard, Thorsten Holz, and Christian Rossow. Leveraging semantic signatures for bug search in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014.

[121] Ashkan Rahimian, Paria Shirani, Saed Alrbaee, Lingyu Wang, and Mourad Debbabi. Bincomp : A stratified approach to compiler provenance attribution. *Digital Investigation*, 14 :S146 – S155, 2015.

[122] Kimberly Redmond, Lannan Luo, and Qiang Zeng. A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis. *CoRR*, abs/1812.09652, 2018.

[123] Pau Riba, Andreas Fischer, Josep Lladós, and Alicia Fornés. Learning graph edit distance by graph neural networks. *Pattern Recognition*, 2021.

[124] Nathan Rosenblum, Barton Miller, and Xiaojin Zhu. Extracting compiler provenance from program binaries. *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 21–28, 2010.

[125] Nathan Rosenblum, Barton P. Miller, and Xiaojin Zhu. Recovering the toolchain provenance of binary code. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2011.

[126] Nathan Rosenblum, Barton P. Miller, and Xiaojin Zhu. Recovering the toolchain provenance of binary code. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, page 100–110, 2011.

[127] Peter J. Rousseeuw. Silhouettes : A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20 :53–65, 1987.

[128] Arthur L Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3) :210–229, 1959.

[129] Alberto Sanfeliu and King-Sun Fu. A distance measure between attributed relational graphs for pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, 1983.

[130] Igor Santos, Felix Brezo, Javier Nieves, Yoseba K. Penya, Borja Sanz, Carlos Laorden, and Pablo G. Bringas. Idea : Opcode-sequence-based malware detection. In Fabio Massacci, Dan Wallach, and Nicola Zannone, editors, *Engineering Secure Software and Systems*, 2010.

[131] Igor Santos, Javier Nieves, and Pablo G. Bringas. Semi-supervised learning for unknown malware detection. In Ajith Abraham, Juan M. Corchado, Sara Rodríguez González, and Juan F. De Paz Santana, editors, *International Symposium on Distributed Computing and Artificial Intelligence*, pages 415–422, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[132] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1) :61–80, 2008. Publisher : IEEE.

[133] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. Disassembly of executable code revisited. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, pages 45–54. IEEE, 2002.

[134] Francesc Serratosa. Fast computation of bipartite graph matching. *Pattern Recognition Letters*, 2014.

[135] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(77) :2539–2561, 2011.

[136] Paria Shirani, Lingyu Wang, and Mourad Debbabi. Binshape : Scalable and robust binary library function identification using function shape. In *DIMVA*, pages 301–324. Springer Publishing, 2017.

[137] Carlos Silla and Alex Freitas. A survey of hierarchical classification across different application domains. *Data Mining and Knowledge Discovery*, 22 :31–72, 01 2011.

[138] P.K. Singh and A. Lakhotia. Static verification of worm and virus behavior in binary executables using model checking. In *IEEE Systems, Man and Cybernetics SocietyInformation Assurance Workshop, 2003.*, pages 298–300, 2003.

[139] Karen Sparck Jones. *A Statistical Interpretation of Term Specificity and Its Application in Retrieval*, page 132–142. Taylor Graham Publishing, GBR, 1988.

[140] W. Tang, P. Luo, J. Fu, and D. Zhang. Libdx : A cross-platform and accurate system to detect third-party libraries in binary code. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 104–115. IEEE Computer Society, 2020.

[141] Wei Tang, Yanlin Wang, Hongyu Zhang, Shi Han, Ping Luo, and Dongmei Zhang. Libdb : An effective and efficient framework for detecting third-party libraries in binaries. In *Proceedings of the 19th International Conference on Mining Software Repositories*, New York, NY, USA, 2022. Association for Computing Machinery.

[142] Radare2 Team. Radare2 book. In *GitHub*. 2017.

[143] Zhenzhou Tian, Yaqian Huang, Borun Xie, Yanping Chen, Lingwei Chen, and Dinghao Wu. Fine-grained compiler identification with sequence-oriented neural modeling. *IEEE Access*, 9 :49160–49175, 2021.

[144] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[145] A. Viet Phan, M. Le Nguyen, and L. Thu Bui. Convolutional neural networks over control flow graphs for software defect prediction. In *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 45–52, 2017.

[146] L. Vinciguerra, L. Wills, N. Kejriwal, P. Martino, and R. Vinciguerra. An experimentation framework for evaluating disassembly and decompilation tools for c++ and java. In *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings.*, pages 14–23, 2003.

[147] Vijay Walunj, Gharib Gharibi, Duy H. Ho, and Yugyung Lee. Graphevo : Characterizing and understanding software evolution using call graphs. In *2019 IEEE International Conference on Big Data (Big Data)*, 2019.

[148] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. Jtrans : Jump-aware transformer for binary code similarity detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, page 1–13, New York, NY, USA, 2022. Association for Computing Machinery.

[149] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM ASE*, pages 87–98, 2016.

[150] Richard C. Wilson and Ping Zhu. A study of graph spectra for comparing graphs and trees. *Pattern Recognition*, 2008.

[151] Wolfgang Wögerer. A survey of static program analysis techniques. Technical report, Citeseer, 2005.

[152] Ming Xu, Lingfei Wu, Shuhui Qi, Jian Xu, Haiping Zhang, Yizhi Ren, and Ning Zheng. A similarity metric method of obfuscated malware using function-call graph. *Journal of Computer Virology and Hacking Techniques*, 2013.

[153] Rui Xu and Donald Wunsch. Survey of clustering algorithms. *IEEE Transactions on neural networks*, 16(3) :645–678, 2005.

[154] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[155] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. Spain : Security patch analysis for binaries towards understanding the pain and pills. In *IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017.

[156] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012.

[157] Jiaqi Yan, Guanhua Yan, and Dong Jin. Classifying malware represented as control flow graphs using deep graph convolutional neural network. In *49th Annual IEEE/IFIP DSN*, pages 52–63, 2019.

[158] Jia Yang, Cai Fu, Xiao-Yang Liu, Heng Yin, and Pan Zhou. Codee : A tensor embedding scheme for binary code search. *IEEE Transactions on Software Engineering*, 2021.

[159] S. Yang, Z. Shi, G. Zhang, M. Li, Y. Ma, and L. Sun. Understand code style : Efficient cnn-based compiler optimization recognition system. In *ICC 2019 - IEEE*, pages 1–6, 2019.

[160] Zhiping Zeng, Anthony K. H. Tung, Jianyong Wang, Jianhua Feng, and Lizhu Zhou. Comparing stars : On approximating graph edit distance. *Proc. VLDB Endow.*, 2(1) :25–36, aug 2009.

[161] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. An end-to-end deep learning architecture for graph classification. In *32th AAAI Conference on Artificial Intelligence*, 2018.

[162] Wenting Zhao, Chunyan Xu, Zhen Cui, Tong Zhang, Jiatao Jiang, Zhenyu Zhang, and Jian Yang. When work matters : Transforming classical network structures to graph cnn. *arXiv :1807.02653*, 2018.

[163] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks : A review of methods and applications, 2018.

Table des figures

Liste des tableaux

Résumé

Dans le domaine du génie logiciel, assurer la qualité et la sûreté des logiciels est complexe. Ce contexte est dû à un ensemble de facteurs, notamment l'utilisation croissante de bibliothèques et le recours à des pratiques comme la copie de codes à partir de services en ligne. Une réponse courante à cette problématique est l'application de méthodes formelles de validation des programmes avant leur diffusion. Cette approche, cependant, requiert une compréhension précise des enjeux à vérifier et un haut degré d'expertise. Cette thèse introduit des méthodes innovantes de rétro-ingénierie pour collecter automatiquement des informations sur l'origine d'un programme et pour identifier des clones de programmes au sein de larges jeux de données. Notre première contribution est le nouveau modèle de réseau de neurones Site Neural Network (SNN) qui prédit la chaîne de compilation utilisée pour produire un programme entier. SNN offre une grande rapidité ainsi qu'une bonne précision. Sa modularité grâce à l'utilisation de hiérarchies de classificateurs permet de considérer facilement des chaînes de compilation supplémentaires. Notre seconde contribution est Program Spectral Similarity (PSS), un outil qui fournit un moyen rapide et efficace de détecter des clones de programmes, même quand leur architecture matérielle visée diffère ou en cas d'offuscation. Contrairement aux méthodes basées sur les fonctions binaires ou sur la distance d'édition des graphes, qui sont chronophages et peu robustes, PSS s'appuie sur l'analyse spectrale de graphes pour mesurer la similarité entre programmes. Cette thèse participe ainsi à renforcer la sécurité des systèmes en mettant à disposition des outils pour identifier rapidement les clones de programmes malveillants. En outre, elle apporte un soutien à l'investigation numérique en donnant des informations pertinentes sur la chaîne de compilation. Ce travail ouvre la voie à de nouveaux réseaux de neurones spécialisés pour les programmes, ainsi qu'au développement de méthodes d'analyse spectrale pour l'étude de la similarité des codes binaires.

Mots-clés: génie logiciel, sécurité informatique, recherche de clones, analyse spectrale, chaîne de compilation, apprentissage automatique

Abstract

In the field of software engineering, ensuring the quality and security of software is complex. This context is due to a set of factors, notably the increasing use of libraries and the use of practices such as copying codes from online services. The usual solution to this problem is the application of formal methods for program validation before their release. However, this approach requires a precise specification and a high degree of expertise. This thesis introduces new reverse engineering methods to automatically collect information about a program toolchain provenance and identify program clones within large data repositories. Our first contribution is the innovative neural network model Site Neural Network (SNN), which predicts the compilation toolchain used to produce an entire program. SNN offers excellent speed as well as good accuracy. Its modularity due to the use of hierarchies of classifiers allows for easy consideration of additional toolchains. Our second contribution is the Program Spectral Similarity (PSS), a tool that provides a quick and efficient way to detect program clones, even when their target hardware architecture differs or in the case of obfuscation. Unlike binary function-based methods or graph edit distance methods, which are time-consuming and low resilient, PSS relies on the spectral analysis of graphs to measure the similarity between programs. This thesis thus contributes to cyber security by providing tools to identify malware clones quickly. In addition, it supports computer forensics by providing relevant information on the compilation chain. This work paves the way for new neural networks for programs, as well as the development of spectral graph analysis methods for studying binary code similarity.

Keywords: software engineering, cyber security, clone search, spectral analysis, toolchain provenance, graph neural networks