YaDiff

Benoît Amiaux, Jérémy Bouetard, Valérian Comiti, Frédéric Grelot, Eric Renault et Martin Tourneboeuf
Benoît.Amiaux@intradef.gouv.fr
Jérémy.Bouetard@intradef.gouv.fr
Valérian.Comiti@intradef.gouv.fr
Frédéric.Grelot@intradef.gouv.fr
Eric.Renault@intradef.gouv.fr
Martin.Tourneboeuf@intradef.gouv.fr

Direction Générale de l'Armement

Résumé. YaDiff est un outil développé au sein des laboratoires de rétro-ingénierie de DGA-MI pour permettre facilement la propagation d'information d'une base IDA [11] à l'autre.
Nous présentons l'état de l'art des méthodes existantes et leurs limites : fermeture du code, difficulté d'utilisation, performances, incapacité à analyser des bases de taille importante.
Nous présentons ensuite une première méthode « empirique » développée pour répondre à ces problèmes : elle allie simplicité d'utilisation et fonctionnalités étendues. L'outil développé permet de faire une analyse comparative de deux bases afin de propager un maximum d'information d'une base à l'autre (par exemple, bibliothèque avec les symboles vs bibliothèque embarquée en statique dans un binaire sans symbole).
Par informations, nous entendons les données suivantes : noms de fonctions, de données globales, de labels, commentaires, prototypes de fonctions ou données, renommage (variables, registres...), typage de structures...
Nous présentons enfin une méthode utilisant des techniques d'intelligence artificielle ayant pour but de pallier aux principales limites de cet algorithme et de ceux existants : incapacité à identifier les fonctions semblables si elles ont trop changé, mais également incapacité à fonctionner entre deux architectures (32 vs 64 bits, ou ARM vs x86 par exemple). Nous présentons les résultats obtenus avec cette méthode et comment ils sont intégrés à YaDiff.
YaDiff est un outil de la suite des « YaTools » (avec YaCo, présenté au SSTIC 2017).

1 Introduction

Le projet YaDiff a vu le jour suite au problème récurrent du suivi de version lors de la rétro-ingénierie de binaires sous IDA [11]. En effet,

quel rétro-concepteur n'a jamais été confronté, après de longues heures d'analyse d'un binaire, à la difficulté de devoir transposer le fruit de son travail sur la toute dernière version dudit binaire ?

Fort heureusement, ce problème a déjà suscité de nombreux travaux et outils permettant d'apporter une réponse partielle. On citera par exemple BinDiff [2,3], Diaphora [9] et TurboDiff [1] parmi les outils les plus connus. Toutefois, aucune de ces solutions actuelles ne permet de répondre aux contraintes auxquelles nous avons été confrontés, à savoir :

- **analyser des binaires complexes** : la plupart des outils de comparaison échouent à traiter des binaires de taille conséquente ;
- **avoir un temps d'exécution raisonnable** : un temps d'exécution qui se compte en heures, voire en journées rend l'utilisation d'un outil rédhibitoire qui finit alors par ne plus être utilisé ;
- **propager le maximum d'information de la base précédente vers la nouvelle base** : lorsque deux fonctions sont identiques entre deux binaires, pourquoi se limiter à propager uniquement le nom de la fonction ? Nous souhaitons en effet propager également les labels, les commentaires, les prototypes, voire même les renommages de registres ou les typages de structures ! De même, si une fonction a changé mais que certains blocs basiques sont correctement identifiés comme identiques, nous souhaitons également propager les informations internes à ces blocs ;
- **gérer plusieurs architectures** : dans la mesure du possible, il faut pouvoir gérer une propagation multi-architecture, supporter les changements d'options de compilation et de système d'exploitation. Par exemple, les binaires compilés pour Linux possèdent souvent des symboles embarqués qu'il serait utile de propager vers des binaires Windows sans symboles ;
- **propager les informations d'une bibliothèque dans l'analyse d'un produit** : il arrive fréquemment qu'un produit compilé contienne une bibliothèque connue (OpenSSL, libc...) et incluse en statique. L'idéal est alors de compiler cette bibliothèque dans la même version et avec des options de compilation proches, pour ensuite propager ses symboles vers le binaire étudié.

Cet article s'articule en trois parties permettant de mettre en avant les apports du projet YaDiff aussi bien en ce qui concerne son approche que ses résultats. Une première partie présente un état de l'art afin de situer YaDiff dans le microcosme de la *différenciation* de binaires. Dans une deuxième partie, le fonctionnement de YaDiff, dit « legacy », est détaillé à

travers l'enchaînement des algorithmes implémentés. Enfin, une troisième partie aborde ce problème sous l'angle de l'« Intelligence Artificielle » pour apporter de nouvelles perpectives dans le domaine.

Remarque : YaDiff, ainsi que la majorité de nos travaux de rétro-ingénierie, est basé sur IDA de Hex-Rays. C'est un outil qui, malgré ses nombreux défauts, est encore aujourd'hui le plus efficace pour nous que ce soit en terme d'interface utilisateur ou puissance d'analyse. Néanmoins, l'architecture de YaDiff permet d'envisager des transferts vers d'autres outils tel que Binary Ninja ou Radare2 : il suffirait en effet d'y implémenter les fonctions d'import et d'export, par ailleurs communes à YaCo.

1.1 État de l'art

Méthodes de comparaisons. La figure 1 présente les relations entre les différents algorithmes de comparaison de chaînes de caractères, graphes, vecteurs ou binaires. On remarque immédiatement que la problématique de comparaison d'objets complexes n'est pas récente, et n'est pas limitée au champ de l'analyse de binaires : la physique, mais également la biologie ont d'énormes besoins dans ce domaine. Il apparait également que, ces dernières années, de nombreuses solutions ont vu le jour pour tenter de résoudre ce problème, mais nous montrerons en quoi ces solutions sont insatisfaisantes en l'état.

Comparaison de binaires. De nombreux outils proposent d'effectuer de la correspondance de fonctions entre deux binaires différents. Nous pouvons en dénombrer une vingtaine : l'objectif de cette section n'est donc pas de les décrire tous. Nous les regrouperons par catégorie :

- **produits commerciaux « tout-en-un » :** il existe des kits d'outils commerciaux. Ces outils possèdent quelques heuristiques, parfois intéressantes, souvent basées sur des signatures de chaînes[1]. Ils ont le désavantage d'être peu populaires, peu documentés, propriétaires et payants. Nous ne les avons pas testés ;
- **produit à base de « signatures » :** d'autres produits utilisent des algorithmes permettant de signer des fonctions pour ensuite les comparer[2]. La forme des signatures dépend de l'algorithme, par exemple Fcatalog [14] utilise une liste de n-grams, Gorille [5] une liste de parcours d'arbre et BinDiff [5] un vecteur d'entiers ;

[1] PeHash, WinHex, Relyze, eEye, BitBlaze [13]

[2] BinDiff [3], Diaphora [9], TurboDiff [1], PatchDiff [12], DarunGrim [7], RaDiff [10], Fcatalog [14], BinSequence [6] Gorille [4, 5]

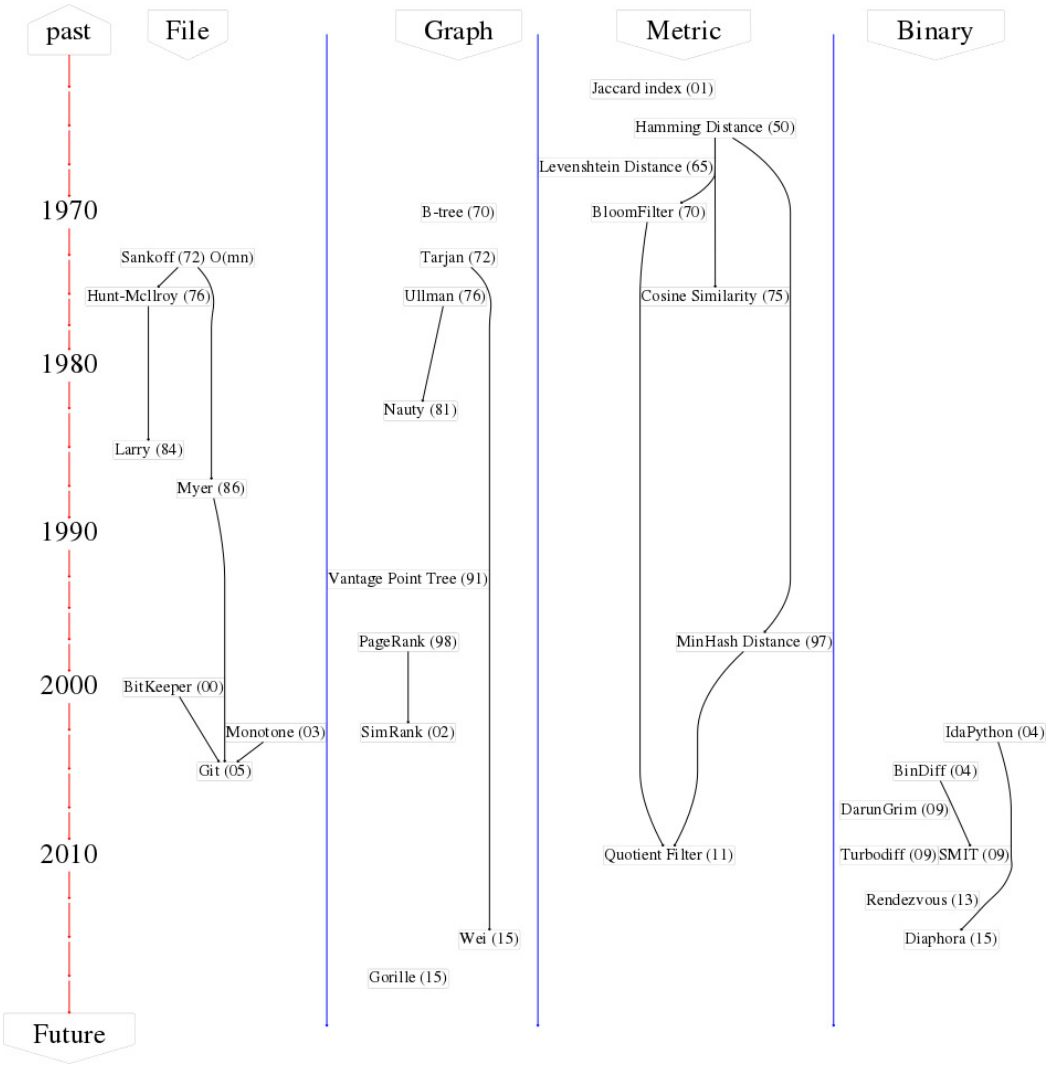


Fig. 1. Algorithmes de comparaison

– **produit d'analyse « dynamique » :** plusieurs algorithmes de comparaison comportementale existent, par exemple en analysant la distribution des appels systèmes[3]. Ce type de mécanismes se retrouve entre autres dans les antivirus. Ils ne nous sont cependant d'aucun intérêt car ils ne peuvent faire une comparaison suffisamment fine, entre fonctions, et ne peuvent comparer que les binaires entiers, voir les produits.

Utilisation de signatures. Beaucoup de logiciels de comparaison signent leurs objets, et travaillent à différents niveaux : les fonctions (BinDiff [3], Diaphora [9], TurboDiff [1]), les blocs basiques (BinDiff) ou les chemins dans le graphe de flot de contrôle (Gorille [5]).

Ces logiciels utilisent des signatures où chaque champ représente une caractéristique de l'objet : on peut représenter ces signatures comme des vecteurs, sur lesquels l'outil définit une distance qui lui permettra d'effectuer des comparaisons entre fonctions. Ce type de signatures nous intéresse car elles peuvent s'utiliser conjointement. En effet, un méta algorithme peut combiner les résultats de plusieurs algorithmes de signature par concaténation de vecteurs, et utiliser une nouvelle mesure de distance (en se basant sur le principe de l'inégalité triangulaire, non détaillé ici, qui permet de définir une mesure de distance sur le vecteur complet en se basant sur les mesures définies sur des parties du vecteur), diminuant ainsi le taux de faux-positifs et faux-négatifs. À titre d'exemple, TurboDiff ne prend en compte que :

– la forme du graphe de flot de contrôle ;
– le condensat de chaque bloc d'instructions ;
– le nombre d'instructions de chaque bloc.

La simplicité des signatures et l'utilisation de code natif permettent à TurboDiff d'être le plus rapide des comparateurs *monothreadés*. Ce principe nous a inspiré, d'autant que le code est en source libre. On retrouvera ainsi dans YaDiff des heuristiques similaires, combinées à d'autres qui permettent de diminuer le nombre de faux négatifs.

À notre connaissance, aucun outil n'a tenté de mettre en oeuvre un algorithme d'apprentissage automatique (plus vulgairement appelé Intelligence Artificielle) dans le calcul de la distance entre deux fonctions.

Par ailleurs, si l'algorithme de signature initial de YaDiff est relativement simple, nous présenterons les méthodes de propagation « de

[3] BinSim [8]

proche en proche » implémentées dans YaDiff. Nous ne connaissons aucun autre outil utilisant de telles méthodes, et nous montrerons en quoi elles permettent d'obtenir une confiance très importante dans les résultats.

1.2 Apports de YaDiff

Tous les logiciels étudiés :

- utilisent une seule et unique méthode (isomorphisme de sous-arbre, distance euclidienne entre vecteurs, apprentissage supervisé) ;
- ne savent pas gérer les binaires lourds, quand la base IDA est de taille conséquente ;
- ne traitent pas les correspondances : ces outils annoncent des « scores de proximité », il revient alors à l'expert le soin d'exploiter ce résultat de la manière qu'il souhaite ;
- n'utilisent pas YaCo qui permet de propager la sortie et de digérer l'entrée via un module IDA natif et optimisé.

Hormis YaDiff, BinDiff est le seul qui :

- compare plus d'un type d'objet : les fonctions, les blocs basiques et les séquences ;
- possède un mode « sans faux positifs ». Un expert doit pouvoir faire confiance aux résultats, dans le cas contraire, l'outil sera jugé comme improductif et ne sera pas utilisé.

Cependant, BinDiff est un logiciel propriétaire, le code source est fermé rendant impossible l'amélioration des résultats ou l'ajout de fonctionnalités. À l'inverse, YaDiff :

- possède plusieurs méthodes de corrélation, trois actuellement ;
- utilise IDA qui est excellent dans l'identification des fonctions isolées ;
- exploite pleinement les résultats en créant une nouvelle base IDA issue de la fusion entre le binaire en cours d'analyse et un binaire déjà documenté ;
- fonctionne aussi bien au niveau bloc d'instructions que fonction : il ira jusqu'à propager un commentaire au milieu d'un bloc d'une fonction dans laquelle d'autres blocs auront été modifiés ;
- effectue la corrélation également sur les objets de données : constantes, champs statiques, chaînes de caractères...
- fonctionne de manière efficace, y compris sur des bases de l'ordre du gigaoctet non-compressé.

2 Orchestrateur d'algorithmes : YaDiff « legacy »

Nous avons fait le choix de mettre au point plusieurs algorithmes indépendants plutôt qu'un seul algorithme complexe. Les algorithmes « d'association initiale » sont exécutés en premier, puis les algorithmes dits « de propagation » sont alors exécutés par alternance : tant qu'un algorithme apporte des nouvelles relations, les autres algorithmes sont réexécutés afin de profiter de ces relations et continuer la propagation.

Tous les algorithmes sont également paramétrables, ce qui permet d'ajuster la confiance souhaitée dans les résultats. Ils pourront ainsi, soit tenter de propager le plus d'informations possible en générant peut-être des faux positifs, soit être très stricts dans les associations, et n'associer que des objets lorsque les correspondances sont certaines. Les trois algorithmes développés se complètent ainsi efficacement.

2.1 Définitions

Afin de lever toute ambiguïté, nous apportons les définitions fondamentales nécessaires pour la suite de cet article.

Objet : tout ce que l'on va essayer de faire correspondre dans les bases : les fonctions, les blocs basiques, les éléments de données (chaîne de caractères, constante, zone mémoire), voire les structures sont considérés de manière équivalente sous cette terminologie.

Bloc basique : ce terme est issu de la terminologie utilisée par le désassembleur IDA (« basic block » en anglais). Un bloc basique est une séquence d'instructions successives au sens de l'adressage, comportant un seul point d'entrée et un seul point de sortie.

Élément de donnée : objet se trouvant généralement dans les sections data et rodata, non exécutable, et contenant des données accessibles depuis le code du programme.

Fonction : ensemble de blocs basiques et des liens qui les relient pour en faire un graphe comprenant un (exceptionnellement plusieurs) point d'entrée, et un ou plusieurs points de sortie.

Référence croisée : lien logique entre deux objets. Correspond globalement à la terminologie « Xref » dans IDA. Peut relier un bloc basique et une fonction (appel), une fonction[4] vers un bloc basique, un bloc basique vers une donnée (lecture, écriture, chargement d'adresse), mais également un bloc basique vers un champ de structure (application d'un type « champ de structure » sur une opérande).

[4] Dans le modèle YaCo, une fonction est un objet vide qui possède des références croisées vers les objets de type bloc basique qui la composent.

```
function_example:
lea  eax, [ds:some_structure_value]
mov  ebx, [eax+the_structure.field_at_offset_10]
call other_function
ret
```

Dans l'exemple ci-dessus, la fonction est constituée d'un seul bloc basique, contenant une référence croisée descendante vers un élément de données (`some_structure_value`), une vers un champ de la structure `the_structure`, et enfin une vers la fonction `other_function`. Inversement, la fonction `other_function` possède une référence croisée montante vers le bloc basique (ainsi que vers tous les autres blocs basiques qui l'appellent ou la référencent).

2.2 Signatures

Suivant les algorithmes utilisés, YaDiff peut avoir besoin de signatures sur des objets et/ou des relations entre ces objets. D'un point de vue mathématique, ces relations peuvent être considérées comme des signatures (la signature d'une fonction est la somme des signatures de ses blocs basiques et de ses relations).

La génération de signatures nécessite une attention particulière. C'est sur elles que vont reposer les algorithmes de correspondance. Elles doivent permettre de discriminer une fonction d'une autre tout en restant tolérantes à certains changements (architecture, options de compilation, patch). Quand un *analyste* propage ses symboles d'un binaire à un autre, il souhaite lier les effets des éléments qu'il a déjà analysés à son nouveau produit.

Idéalement, une même fonction de par ses effets (par exemple « la fonction printf ») implémentée de plusieurs manières différentes (en C, en ADA, en RUST), puis compilée sous diverses architectures et avec diverses optimisations devrait systématiquement être reconnue comme identique.

Nous n'avons évidemment pas résolu ce problème parfaitement, mais il est tout de même important de ne pas oublier cet objectif final : cela permettra de garantir que l'on se concentrera sur des signatures les plus pertinentes possibles, sans ajouter trop de complexité aux algorithmes de correspondance qui les utiliseront.

Nous proposons deux signatures relativement simples pour les instructions. Ces signatures ne sont pas nécessairement originales (voir [3, 9]), mais, couplées aux algorithmes décrits par la suite et étant données nos

contraintes (notamment de performance), elles apparaissent comme tout à fait adaptées à notre problématique.

- **InvariantBytes** : pour chaque instruction, YaDiff demande à IDA les bits invariants de l'instruction. IDA renvoit alors un masque de bit sur l'instruction (dont on connait déjà la taille), qui correspondent aux bits décrivant le mnémonique (`mov`, `add`, `inc`...). Cela induit une certaine flexibilité et permet d'être relativement tolérant aux affectations de registres par le compilateur, mais aussi aux offsets ou adresses présentes dans les opérandes. On concatène ensuite cette suite et un condensat est calculé sur l'ensemble, qui correspond à la signature de l'objet. Nous n'avons pas de contraintes fortes de non-collision sur les condensats (cela peut arriver, mais ne sera pas dramatique), aussi un MD5 répond largement à nos besoins.
- **FirstBytes** : pour chaque instruction, on ne garde que le premier octet. Cet algorithme est particulièrement performant pour les architectures RISC, par exemple ARM (car il est quasiment équivalent au premier, mais ne nécessite pas de désassembler toutes les instructions).

Pour les données initialisées (sections data et rodata), on calcule un condensat MD5 de la donnée. De cette façon une chaîne de caractères produit le même condensat peu importe l'architecture, de même pour les constantes. En revanche, les données non initialisées sont prises en compte avec une signature nulle : Comme nous le verrons par la suite, cela signifie que *toutes* les données non initialisées ont la même signature et entreront en collision : ce sera alors aux algorithmes travaillant par propagation de les discriminer (et, dans ce travail, nous verrons que l'algorithme par propagation descendante est redoutable...).

2.3 Algorithmes

Les algorithmes doivent être faciles à comprendre, simples à mettre en oeuvre et efficaces. Quand un expert souhaite retrouver une fonction dans un autre binaire, il commence généralement par chercher des éléments remarquables (chaînes de caractères, fonctions déjà identifiées, données initialisées). Il utilise ensuite les références croisées pour naviguer de fonction en fonction et accroître sa connaissance du nouveau binaire. YaDiff ne fait rien de plus compliqué mais accélère et automatise ce processus.

Dans les illustrations des exemples suivants, les objets représentés par des ellipses ayant les mêmes condensats possèdent les mêmes signatures.

Algorithme 1 : Association initiale. Avant tout traitement complexe, il est nécessaire d'établir des relations de confiance entre les objets de deux bases. Il faut parvenir à associer deux objets dont on est sûr qu'ils sont équivalents.

Le principal problème auquel il faut faire face se pose pour les objets qui apparaissent plusieurs fois dans une base : un petit bloc basique peut ainsi apparaître plusieurs dizaines de fois dans un seul binaire. Ceci est d'autant plus vrai que la méthode de signature consiste en un condensat sur les bits invariants des instructions (donc en ignorant les bits qui décrivent les valeurs d'opérande).

La méthode est alors assez simple : tout objet dont la signature collisionne avec une signature d'un autre objet de la même base est ignoré. Les objets restants (signature qui n'apparaît qu'une fois) dont les signatures correspondent entre les deux bases sont alors associés avec une confiance maximale. Le résultat donne des relations de type *un-à-un*. La figure 2 montre un exemple d'association initiale.

D'expérience, cette première passe d'association est la plus importante, car elle provoque généralement l'association de près de la moitié des blocs de base des binaires.

Note sur le C++ : en C++, il y a très peu de références croisées entre les fonctions, car beaucoup d'appels passent par les tables virtuelles. En l'occurence, il serait probablement pertinent d'améliorer l'algorithme de signature pour ce cas spécifique, en considérant les tables virtuelles comme des données complexes possédant des références croisées sur chaque fonction virtuelle : dans ce cas, l'algorithme de références croisées descendantes permettrait peut-être de résoudre les conflits. Cette amélioration n'a pas encore été implémentée.

Algorithme 2 : Propagation par références croisées montantes. Une fois que des relations sûres sont établies entre les objets, on peut partir de ces associations pour propager la connaissance et associer d'autres objets.

Pour y parvenir, l'algorithme parcourt les références croisées montantes (ou parentes) pour se propager. Il récupère les signatures des objets parents de chaque objet associé. Si deux objets dans les bases 1 et 2 référencent une paire d'objets dont les signatures correspondent, il est possible d'associer les objets parents. Il est possible de réaliser cette association même si leurs signatures collisionnent avec d'autres signatures de la base (l'algorithme initial l'ayant ignoré). La figure 3 décrit un exemple de propagation montante. De manière indirecte nous avons reconstruit une signature

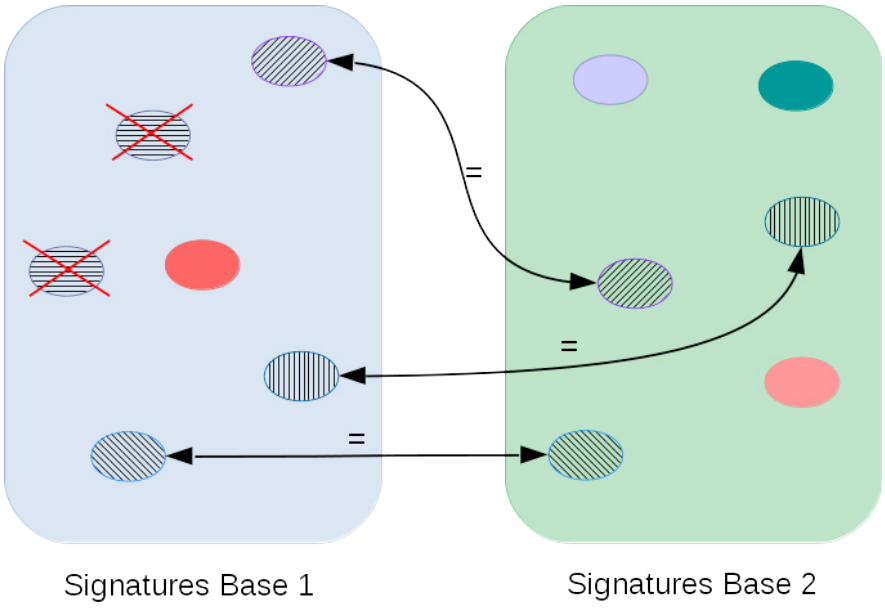


Fig. 2. Association initiale

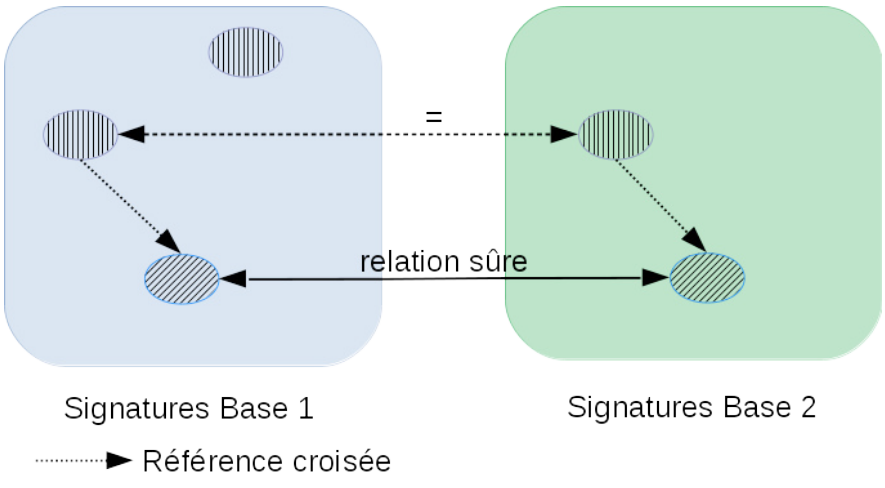


Fig. 3. Association par références croisées montantes

composée de la signature des objets et des références croisées de cet objet. La référence croisée est une caractéristique qui permet de discriminer un objet d'un autre.

La figure 4 montre la détection de nouvelles références sur le premier objet dû à un nouvel objet ou à un objet modifié.

Dans le cas où il y a uniquement une seule référence non résolue de chaque côté, il est quand même possible d'associer ces objets. Mais cette fois-ci, les signatures étant différentes, les objets ne sont plus les mêmes : il s'agit d'association d'objets qui diffèrent entre les deux versions. La figure 5 décrit ce principe.

Cette méthode permet d'associer des fonctions dans des architectures différentes, car les signatures ne sont pas identiques.

Par exemple, nous avons constaté qu'en utilisant uniquement les références croisées sur les chaînes de caractères exploitées dans l'algorithme 1 d'association initiale (condensat identique), il est possible d'associer les fonctions qui y font référence. Elles seront identifiées comme sémantiquement équivalentes mais d'implémentations différentes. Cette méthode montre de nombreuses limites mais, en l'absence de variante plus performante, permet déjà de donner de bons points d'accroche à l'analyste qui s'intéresse à une nouvelle architecture.

Algorithme 3 : Propagation par références croisées descendantes. De manière similaire aux références croisées montantes, on utilise les relations sûres comme entrée de l'algorithme. Au lieu de prendre les références croisées montantes, on parcourt les références croisées descendantes (ou enfants). Cette fois-ci, contrairement à l'algorithme 2, on utilise également l'offset de la référence croisée dans l'objet. Il est ainsi possible d'associer des objets ayant les mêmes signatures à des offsets différents. Cette méthode permet également de détecter des objets différents dans le cas où aux mêmes offsets les signatures des objets référencés sont différentes. La figure 6 montre un exemple d'associations en utilisant les références croisées descendantes et leurs offsets.

2.4 Résultats et applications

Les résultats présentés ci-dessous ont été réalisés en appliquant YaDiff sur un binaire de taille conséquente, pour à la fois montrer les résultats de l'outil, mais également sa rapidité d'exécution. Nous avons ainsi sélectionné le logiciel *Chromium* comme binaire de test que nous avons extrait des dépôts officiels Debian. Ce choix revêt un intérêt particulier, car il donne

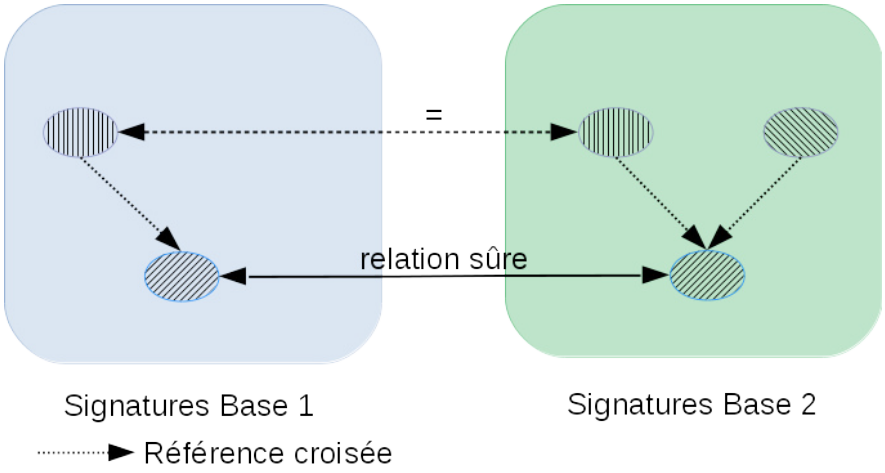


Fig. 4. Nouvelle référence

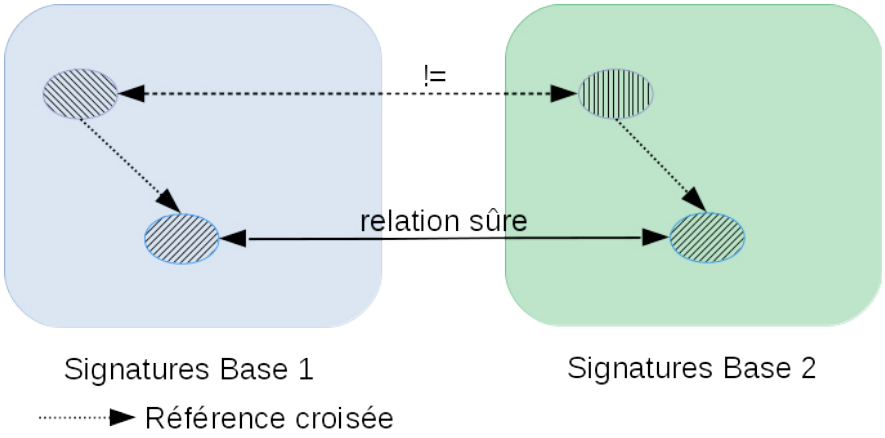


Fig. 5. Détection d'objet modifié

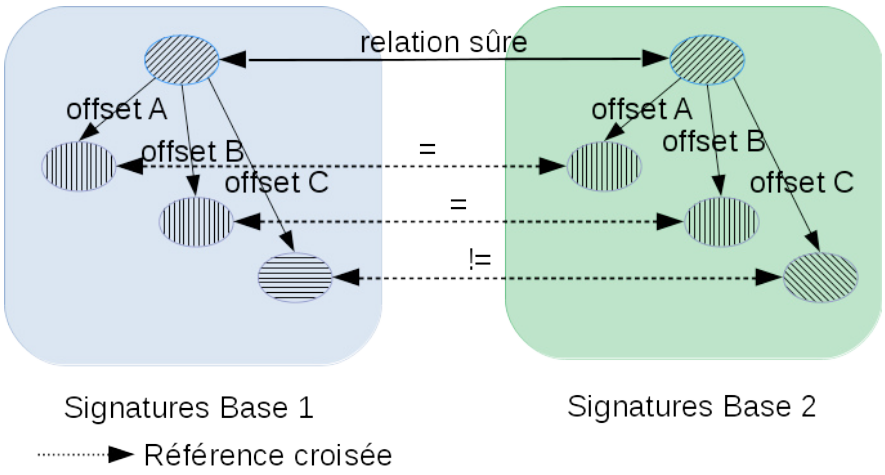


Fig. 6. Association par références croisées descendantes

également un exemple de propagation d'informations entre un binaire dont on a les symboles et de nombreuses informations de débogage (Chromium) et un binaire propriétaire, potentiellement sans symbole (Chrome). Il s'agit également d'un cas moins favorable pour les algorithmes de YaDiff car il contient beaucoup de code C++ (cf. remarque au paragraphe 2.3). Les versions analysées sont les suivantes :

- `chromium_66.0.3359.22-3_amd64.deb`
- `chromium_65.0.3325.146-4_amd64.deb`
- `chromium_65.0.3325.146-4_arm64.deb`

L'exécution de YaDiff se déroule en plusieurs étapes :

1. Analyser les binaires source et destination avec IDA
2. Exporter les bases IDA vers des bases YaCo (.yadb)
3. Lancer les différents algorithmes de correspondance et exporter les modifications dans une troisième base YaCo
4. Importer ces informations propagées dans la base destination IDA

Les résultats sont présentés dans le tableau ci-dessous. Le nombre de fonctions correspond au nombre minimal de fonctions dans les binaires source et destination.

Versions comparées (AMD64)	`v65` ⇒ `v65`	`v65` ⇒ `v66`
Nombre de fonctions		
total	479 861	479 861
identiques retrouvés	394 994	337 913
différentes retrouvées	0	24 872
Indice de correspondance	82,3 %	75,6 %

Le premier test consiste à exécuter YaDiff sur le même binaire pour identifier sa capacité maximale. En théorie, 100 % des fonctions devraient être retrouvées, et aucune fonction différente ne devrait être identifiée. L'indice de correspondance pour ce test est 82,3 %. Cela peut paraître assez faible mais YaDiff évite tout faux positif : Les fonctions qui n'ont pas pu être associées sont des fonctions qui se ressemblent (mêmes signatures) et qui ont très peu de références croisées, ou qui se trouvent dans les chemins C++ encore difficiles à traiter.

Le second test porte sur des binaires de différentes versions. YaDiff identifie des fonctions qui ont changé, leurs implémentations sont différentes et dans la plupart des cas leurs sémantiques sont équivalentes. YaDiff pourrait repartir de ces associations pour à nouveau se propager

aux parents mais nous avons volontairement empêché ce comportement pour éviter d'induire tout faux positif.

Le facteur temps est pour nous très important, l'exemple de Chromium est donc tout à fait pertinent car la base IDA fait alors de l'ordre de 1.1 Go (pour un binaire de ~120 Mo). Nous nous sommes placés dans la pire situation, où l'indice de correspondance est le meilleur et la quantité d'informations à propager est maximale ; nous avons donc testé le temps de propagation d'une base sur elle-même. Les tests ont été réalisés sur un ordinateur portable classique, avec un processeur i7 3520, 4 Go de RAM, et un unique disque dur à plateaux.

Le temps d'exécution de propagation de `v65` AMD64 vers `v65` AMD64 se décompose de la manière suivante :

Chargement initiale du binaire dans IDA	~1h
Export YAFB de la base source	11m11s
Export YAFB de la base destination	9m59s
YaDiff	4m09s
Import dans IDA	1h07m04s

Nous avons essayé de réaliser ce test avec *Diaphora*. La première phase d'export a duré environ huit heures. Nous avons malheureusement interrompu le traitement suivant, faute de résultats, après 24 heures de calcul. Il est à noter que même avec des binaires de plus petite taille (~50 Mo) nous n'avons pas pu mener le test jusqu'à son terme.

2.5 Conclusion intermédiaire

YaDiff est en l'état utilisable et répond à notre besoin : il est simple d'utilisation et performant y compris sur des binaires de taille importante sur lesquels tous les autres algorithmes échouent à produire le moindre résultat. Nous n'avons pas la prétention de révolutionner les méthodes de correspondance et de différenciation de binaires avec les algorithmes présentés ci-dessus. Notre volonté s'est portée sur la facilité d'utilisation, la confiance et la rapidité des résultats, la quantité des informations propagées, ainsi que l'ouverture du code. Ces objectifs nous semblent atteints. Nous espérons que la diffusion de cet outil vers la communauté facilitera l'introduction d'algorithmes innovants et précurseurs avec toute la puissance des outils YaCo.

Limites. Aujourd'hui, nous pouvons considérer les versions offusquées ou inlinées d'une fonction comme des fonctions différentes. Pour ce qui

est de l'offuscation, il peut arriver qu'une fonction soit coupée en deux ou reçoive un nombre différent de paramètres, ce qui peut être très difficile à analyser.

En ce qui concerne les fonctions inlinées, elles entrent dans l'empreinte des fonctions qui les appellent et n'apparaissent alors plus comme des fonctions à proprement parler dans le binaire. Les détecter est inutile dans le cadre de la propagation des symboles, mais dans l'idéal il faudrait qu'elles ne perturbent pas la détection des fonctions au sein desquelles elles sont inlinées.

Quant aux boucles aplaties, elles changent profondément la structure de la fonction. Dans un premier temps, il sera plus simple de considérer ces fonctions qui ont aplati du code comme des fonctions différentes de celles qui ne l'ont pas fait.

Pour l'expert en rétro-ingénierie, l'idéal serait de retrouver ces fonctions sous le même nom : la route est longue, mais les algorithmes que nous proposons s'y rapprochent.

3 Algorithme d'Intelligence Artificielle

3.1 Introduction

Comme indiqué précédemment, l'outil YaDiff basé sur des algorithmes empiriques remplit très bien l'objectif pour lequel il a été développé : propager d'une base à l'autre des informations documentées par l'expert (ou au travers d'une compilation avec symboles), en apportant une confiance importante dans le fait que les informations sont exactes.

Cependant, les nouvelles technologies basées sur l'apprentissage automatique nous laissent penser qu'il est possible de faire mieux, ou tout du moins de se rapprocher de ce que fait l'expert manuellement : parvenir, avec une vue plus globale, à déterminer lorsque deux fonctions représentent deux version d'un seul et unique code. L'analyse pourrait en effet être indépendante des options de compilations ou du compilateur utilisé, de l'architecture ou des différentes variantes d'une architecture (32 ou 64 bits, mode thumb, endianness...), voire, idéalement, de l'utilisation d'un mécanisme d'offuscation.

Nous allons ainsi vous présenter nos travaux de recherche, en cours d'intégration dans la version opérationnelle de YaDiff, qui utilisent ces méthodes pour améliorer les résultats.

Il est important de noter que l'algorithme présenté ici est complémentaires aux autres algorithmes, pour les raisons suivantes :

– il n'est pas capable de faire les associations de données (par exemple des chaînes de caractères), ce que l'algorithme standard fait déja très bien (par calcul de condensat, ce qui semble la méthode la plus efficace) ;
– il permet d'apporter des associations **initiales**, de la même manière que l'algorithme basé sur les signatures des fonctions : les algorithmes de propagation par référence croisées fonctionnent sur la base de ces associations et en profiteront donc, y compris en mode multi-architecture.

3.2 Définition du problème, modélisation des données

La première étape dans le recours à un algorithme d'apprentissage automatique concerne le traitement des données d'entrées. En particulier, il faut commencer par définir un format de données que l'on fournira à notre algorithme (en l'occurence un réseau de neurones).

Dans le cas de l'analyse de binaire, nous choisissons de nous placer au niveau des fonctions : nous allons tenter de comparer deux fonctions, afin d'indiquer si elles sont similaires ou non, et d'indiquer un degré de similarité.

Les fonctions sont des objets de taille variable, ce qui est un point important du problème. En effet, les réseaux de neurones sont relativement efficaces lorsqu'il s'agit de traiter des données de taille fixe, mais plus difficile à maitriser pour ce qui est des tailles variables car il faut alors généralement recourir à des réseaux récurrents.

Nous avons donc deux possibilités : réduire les fonctions à des objets de taille fixe, ou utiliser des réseaux récurrents. Nous avons choisi la première option. Il pourrait être pertinent de creuser la deuxième option moyennant une modélisation beaucoup plus complexe, ce que nous n'avons pas fait car les résultats sont déjà excellents ainsi.

L'objectif du réseau de neurones sera alors de traiter la donnée d'entrée (un vecteur de taille fixe décrivant la fonction), pour fournir un nouveau vecteur (de taille à déterminer). Nous « demanderons » lors de la phase d'apprentissage à ce que lorsque deux fonctions sont semblables (terme que nous définirons), les vecteurs de sortie soient proches (voir confondus), tandis que lorsqu'elles sont différentes, les vecteurs soient éloignés. En phase d'exploitation, nous n'aurons alors plus qu'à calculer les vecteurs de sortie pour toutes les fonctions d'un binaire et comparer les distances entre les vecteurs qui nous intéressent pour en déduire leur proximité.

La description des vecteurs d'entrées fait l'objet de la sous-section suivante.

3.3 Export

Nos données d'entrée sont des fonctions, qui sont des objets complexes et de taille variable. La première étape de la chaîne de traitement consiste donc à extraire de chacune d'entre elles un grand nombre de caractéristiques scalaires. Ces dernières sont représentées sous forme d'entier ou de réel. Nous utilisons pour cette étape la chaîne YaCo : une fois que le binaire est désassemblé par IDA et que l'auto-analyse est terminée, nous exportons les données à l'aide des méthodes d'exportation déjà implémentées par YaCo (souvenez-vous : celui-ci exporte entre autre les travaux de l'analyste sous forme d'arborescence XML suivie par GIT, qui est alors synchronisé chez les collaborateurs). L'énorme avantage de cette option est que YaCo exporte déjà la majorité des données utiles d'une base IDA : les noms bien sûr, mais aussi les commentaires, les informations de typage de structure dans le code, les renommages de registres, les types des données et des fonctions... Par ailleurs, cela permet également de traiter les données en dehors d'IDA, ce qui apporte beaucoup de flexibilité. En l'occurence, nous profitons également de la flexibilité de cette méthode d'export pour utiliser le format binaire YADB, beaucoup plus compact que le XML et plus rapide à traiter.

Une fois cet export effectué, nous utilisons l'outil (indépendant d'IDA) yadbtovector : celui-ci va transformer chaque fonction enregistrée dans le fichier YADB en un vecteur de taille fixe, comme exigé par l'algorithme d'apprentissage automatique que nous avons choisi.

Le fichier de vecteurs (un par fonction) comporte plusieurs séries de coordonnées décrivant :

- les instructions de la fonction ;
- la forme du graphe de flot de contrôle ;
- les fonctions voisines (appelantes et appelées) de la fonction considérée.

Instructions. Les fonctions sont des objets complexes constitués d'un nombre très variable d'instructions (potentiellement non borné). Ces instructions sont donc les données naturelles à prendre en compte. Cependant, il est impossible d'être exhaustif dans leur description dans la mesure où l'on souhaite obtenir un nombre fixe de valeurs, que l'on considère une fonction à deux ou plusieurs milliers d'instructions. Les instructions contenues dans une fonction sont étiquetées puis leur distribution standardisée.

Chaque instruction est étiquetée d'un ou de plusieurs types suivant les opérations qu'elle effectue : lecture/écriture mémoire directe et indirecte, déréférencement de pointeur, saut (in)conditionnel, instruction

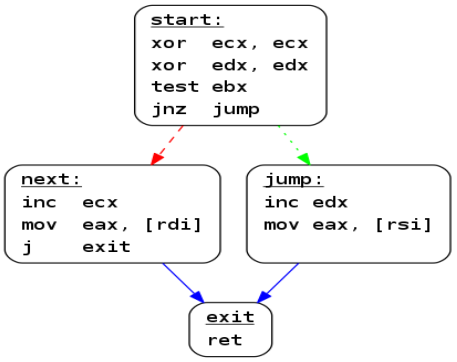
générique (affecté à toute les instructions), appel de fonction, instruction conditionnelle, etc.

Etant donné que les vecteurs doivent être de taille identique quelle que soit la taille de la fonction, nous n'allons pas exporter les informations de chaque instruction mais plutôt de la distribution de chaque type d'instruction au sein d'une fonction : leur nombre, leur position moyenne, l'asymétrie, l'aplatissement, ce qui correspond aux moments centrés de leur distribution.

Il y a d'autres manières de représenter une distribution en un nombre fixe de variables, comme par exemple les quantiles ou les fréquences principales, mais nous avons choisi de les écarter pour l'instant car ces variantes sont moins robustes aux changements. Par exemple, dans une petite fonction l'ajout d'une instruction modifiera toutes les quantiles suivants alors que les moments ne seront que peu affectés. Concernant les fréquences principales et leur amplitude, elles offrent l'avantage d'apporter des informations pertinentes en démasquant les motifs répétitifs : boucles inlinées, appels de fonctions répétitifs, utilisation de templates et de macros. Malheureusement l'extraction de ces fréquences n'a pas encore été implémentée, toujours dans un objectif de simplicité mais également car, comme nous le montrerons, les résultats sont déjà très bons !

Par ailleurs, il ne faut pas oublier que les instructions d'une fonction ne se suivent pas de manière linéaire mais sont présentes dans le binaire sous forme d'un graphe. Les scalaires présentés sont donc calculés sur une version « aplatie » du CFG (Control Flow Graph) dans laquelle la position (offset 0) de chaque bloc basique correspond à sa distance minimale au point d'entrée de la fonction. Les blocs sont ainsi superposés de manière linéaire, puis les moments centrés calculés.

Par exemple, considérons le code suivant et sa version aplatie :



offset	instructions	
0	xor ecx, ecx	
1	xor edx, edx	
2	test ebx	
3	jnz label_1	
4	inc ecx	+ inc edx
5	mov eax, [rdi]	+ mov eax, [rsi]
6	j label_2	+ ret

La position moyenne des déréférencements mémoire sera par exemple la position 5, car il y en a que deux à cet offset. Bien entendu, les moyennes s'entendent de manière pondérées : s'il y avait eu un déréférencement

à l'offset 2 (par exemple « `test [ebx]` »), la moyenne aurait été de $(2 + 5 + 5)/3 = 4$.

Cette première analyse fournit une centaine de valeurs scalaires décrivant les fonctions : elles sont indépendantes de la taille de la fonction, et, hormis les valeurs « nombre d'instruction du type X », elles sont comprises entre 0 et 1.

Graphe de flot de contrôle. Le graphe de flot de contrôle (CFG) d'une fonction est un graphe de blocs basiques. Plus précisément il s'agit d'un arbre orienté enraciné. Il représente l'imbrication et les relations causales des instructions. Nous avons choisi de le représenter indépendamment des instructions présentes dans chacun de ces blocs basiques et avons défini et implémenté le calcul des valeurs suivantes :

- Nombre de blocs basiques
- Nombre d'arêtes
- Nombre de points de retour en arrière (boucles)
- Nombre de croisements (branches)
- Nombre d'intersections en diamant (if ... then ... else)
- Hauteur du graphe minimale et maximale en nombre d'instructions et de blocs basiques
- Largeur maximale : le nombre maximum de basiques blocs à même distance de l'entrée
- Dispersion des tailles des blocs basiques
- Nombre d'instructions pour aller de l'entrée au retour par le chemin le plus court et le plus long

Ces quelques valeurs ont un sens sur la forme du CFG, c'est pourquoi elles ont été sélectionnées. Nous ne faisons que représenter ce qu'un analyste voit dans la vue graphe d'IDA : l'experience nous apprend qu'une manière particulièrement efficace de comparer deux fonction est d'activer cette vue, qui donne instantanément une sorte de signature visuelle très facile à comparer pour un oeil humain, même non expert.

Cette dizaine de variables ne suffit bien sûr pas à décrire ce que l'on voit alors : il sera sans doute pertinent d'ajouter d'autres valeurs. Par exemple nous ne prenons pas en compte l'orientation de l'arbre c'est-à-dire la condition de chaque embranchement. Cela permettrait pourtant de représenter comment une fonction vérifie ses appels.

Il est important de noter que si l'on décidait par erreur d'ajouter des coordonnées qui n'ont pas de sens, on perdrait en temps de calcul bien sûr, mais pas en résultats : l'algorithme finirait par conclure que

ces coordonnées ne sont pas corrélées avec la valeur de sortie, et les écarteraient. Il est donc utile d'ajouter tout ce que l'intuition nous indique comme pertinent. Si le besoin s'en fait sentir, une ultime étape pourra éventuellement consister à analyser la cartographie des caractéristiques (« saliency map » en anglais), qui permet d'analyser le poids de chaque variable d'entrée dans le calcul du score de sortie, et ainsi d'éliminer les caractéristiques inutiles.

Graphe d'appel. La position d'une fonction dans son entourage relativement à ses voisines (appelants/appelés) peut apporter des informations sur son identité. En effet, il arrive très fréquemment qu'un même binaire contienne plusieurs fonctions identiques mais appelées dans des contextes différents : ce sont justement ces fonctions qui posent problème aux algorithmes de signatures (et qui ont mené à la création des algorithmes 2 et 3 dans YaDiff Legacy).

Etant donné que l'on veut également éviter cet écueil dans l'algorithme basé sur l'apprentissage automatique, on a ajouté au vecteur initial, qui correspond à la description des deux paragraphes précédents (appelons le « I ») la moyenne et la médiane des valeurs des I pour toutes les fonctions appelées, appelantes, mais également la déviation par rapport à cette moyenne.

D'une centaine de valeurs de départ, ce nouveau vecteur contient ainsi près de 900 caractéristiques, qui décrivent précisément la fonction considérée mais également ses parents et enfants, ce qui donne une très bonne indication sur son positionnement dans le graphe de flot de contrôle global du programme.

3.4 Distance entre vecteurs de caractéristiques

À ce stade de l'étude, on pourrait commencer à effectuer une première analyse des données, pour voir si une méthode simple permettrait de déterminer les fonctions semblables.

Une méthode naïve pourrait être d'effectuer une simple distance euclidienne[5] entre les vecteurs calculés pour définir la distance. Malheureusement, cela ne sera pas très efficace car on manquera ainsi les relations complexes et, surtout, on donnerait alors autant d'importance à toutes les coordonnées, alors que certaines sont sans doute plus déterminantes (par exemple : le nombre de blocs basiques ou la taille de la fonction).

[5] C'est la racine carrée des carrés des coordonnées de la différence entre deux vecteurs.

Dans un premier temps, il peut être utile de normaliser les vecteurs, et d'effectuer une analyse des composantes principales (PCA) pour restructurer l'espace et ainsi d'organiser les coordonnées selon la dimension de plus forte variance.

Malheureusement, ceci aboutirait alors à une distorsion de l'espace selon ces dimensions, et rien n'indique que la distance induite serait pertinente.

Un partitionnement (« clustering » en anglais) permettrait également de fragmenter l'espace en plusieurs parties. Cependant, avec une base de données qui contient potentiellement des millions de fonctions différentes : une séparation de l'espace en millions de partitions semble inadapté.

Par ailleurs, si l'on n'effectue que des recombinaisons linéaires (du type PCA), on ne pourra jamais mettre en évidence des relations complexes entre les vecteurs, du type : « telle architecture a plus de saut, mais telle autre implémente des instructions conditionnelles », ou une instruction `mov` dans une boucle est potentiellement équivalente à 4 instructions `mov` à la suite... C'est ici que la solution des réseaux de neurones parait plus prometteuse : premièrement, les transformations ne sont pas limitées aux transformations linéaires des entrées, et deuxièmement, on peut espérer que l'algorithme « apprenne » la logique derrière les données et soit capable de plus de généricité.

4 Apprentissage supervisé

4.1 Définition d'un corpus d'apprentissage

Le corpus d'apprentissage doit permettre de fournir l'information à faire apprendre (deux fonctions représentent-elles ou pas la même chose), et être suffisamment fourni car les algorithmes d'apprentissage automatique sont relativement dépendants de la quantité de données en entrée.

Dans un premier temps, nous avons choisi d'extraire tous les fichiers ELFs d'un dépôt Linux Debian Wheezy+Jessie+Stretch. Cela représente environ 400 000 fichiers, pour plus de 50 millions de fonctions. Nous partons alors du principe que deux fonctions qui ont le même nom de fichier binaire et même nom de fonction sont identiques ou proches, sinon elles sont différentes. Cette approximation peut induire des erreurs, mais l'important est qu'elle soit globalement correcte pour que le réseau de neurones apprenne correctement.

Ce corpus d'apprentissage est bien entendu biaisé : les binaires sont majoritairement compilés avec GCC pour Linux, ce qui implique que l'algorithme pourra alors peut-être devenir excellent sur de la comparaison

de binaires linux, mais échouer totalement lorsqu'il s'agira de comparer deux fonctions compilées avec un autre compilateur sur un autre système.

Cependant, une fois la méthode validée, rien n'empêche d'utiliser un corpus plus fourni afin de corriger ce biais, par exemple en y incorporant des DLLs, un autre dépot compilé avec CLANG, ou des dépots compilés avec d'autres options (optimisation, modes d'inlining plus ou moins agressifs...)

La seule contrainte sera alors de disposer de binaires avec les noms de fonctions. On pourrait par exemple constituer un corpus de binaires compilés à la fois pour Linux, Windows, iOS, avec CLANG, GCC, et Visual Studio : des bibliothèques existent avec une telle compatibilité et les utiliser apporterait énormément de diversité à la phase d'apprentissage, induisant une très importante robustesse supplémentaire.

4.2 Extraction de données pertinentes

Les données d'entrée de l'algorithme sont les vecteurs de caractéristiques décrits au chapitre précédent. Pour chaque fonction, on utilise les informations « condensat du binaire », « nom du binaire », « nom de fonction », caractéristiques, architecture (x86, ARM, MIPS, PPC, 32 ou 64 bits, sous forme de bitfield). Bien entendu, le condensat, le nom du binaire et le nom de fonction ne sont pas fournis à l'algorithme d'apprentissage automatique.

4.3 Normalisation des vecteurs

Dans de nombreux domaines, les vecteurs ont des distributions caractéristiques et la normalisation est relativement simple. Dans le cas présent, chaque composante du vecteur représente un domaine particulier, avec sa propre distribution. Nous travaillons aussi parfois sur des métriques non bornées. Par exemple le fait de compter certaines opérations au sein de la fonction peut grandement varier d'une fonction à une autre.

Les composantes d'un vecteur caractéristiques sont par conséquent très hétérogènes. Les distributions étant pour la plupart non naturelles, nous n'avons pas retenu la normalisation standard. Nous avons donc retenu une approche de normalisation mixte linéraire et logarithmique dont le choix pour chacune des composantes dépend de la différence entre la valeur maximale et minimale. Au delà d'un seuil, fixé arbitrairement, la composante est normalisée de manière logarithmique, sinon elle est normalisée linéairement.

Dans le cas linéaire nous avons donc normalisé comme suit :

$$2 * (X - min)/(max - min) - 1$$

Et dans le cas logarithmique :

$$2 * \ln(1 + X - min)/\ln(1 + max - min) - 1$$

4.4 Groupe de fonctions

Nous avons défini au préalable qu'une famille de fonctions au sens de l'entraînement est l'ensemble des fonctions portant le même nom de bibliothèque et le même nom de fonction.

Cependant, nous sommes allés plus loin dans la constitution de nos labels, en prenant en compte les possibilités d'égalité de vecteurs entre familles de fonctions distinctes.

En effet, lorsque des vecteurs normalisés de deux familles de fonctions distinctes sont égaux, on considère que les deux familles appartiennent à ce que nous appelerons par la suite le même groupe de fonctions.

Par exemple, supposons que l'on ait deux binaires A et B, avec deux fonctions chacun, et présents dans deux versions différentes :

- binaire A_v1 : fA_R_1, fA_S_1
- binaire A_v2 : fA_R_2, fA_S_2
- binaire B_v1 : fB_T_1, fB_U_1
- binaire B_v2 : fA_T_2, fB_U_2

De base, on considère qu'il y a quatre familles de fonctions, avec chacune deux versions d'une même fonction :

- fA_R(fA_R_1, fA_R_2)
- fA_S(fA_S_1, fA_S_2)
- fB_T(fB_T_1, fB_T_2)
- fB_U(fB_U_1, fB_U_2)

Nous avons fait l'hypothèse qu'un vecteur décrit de manière pertinente une fonction au dela de son contexte : dans l'idéal par exemple, le vecteur devrait pouvoir nous faire dire qu'« une fonction printf est une fonction qui affiche une chaîne de caractère en se basant sur un format et une suite d'arguments », ce qui est parfaitement indépendant de son implémentation, de l'architecture, des options de compilation, etc.

Supposons ainsi que l'on constate que les vecteurs fA_R_1 et fB_T_2 sont égaux (même si les binaires A et B n'ont rien à voir) : on doit alors pouvoir conclure que les fonctions des familles fA_R et fB_T appartiennent à la même famille. Il reste ainsi trois familles de fonctions identiques :

– f0 (fA_R_1, fA_R_2, fB_T_1, fB_T_2)
– fA_Y(fA_S_1, fA_S_2)
– fB_Y(fB_U_1, fB_U_2)

Nous avons alors créé un dictionnaire de vecteurs afin de tester leur égalité. D'une part, ceci nous a permis de regrouper les familles de fonctions au sein de groupes. D'autre part nous avons pu ainsi éliminer les doublons pour ne conserver pour chaque groupe que des vecteurs uniques, ce qui a son importance comme nous pourrons le voir lors de la création des paires d'entraînement.

Par ce jeu d'égalités successives, des chaînes allant jusqu'à 25 000 familles de fonctions ont été découvertes. Dans le cadre de l'entraînement, nous éliminons tous les groupes dont le nombre de vecteurs uniques dépasse d'un facteur de plusieurs déviations standard l'ensemble des autres groupes. Le facteur actuellement retenu est de 4.0.

4.5 Création d'un jeu de validation

Dans un premier temps, nous avons choisi d'écarter des bibliothèques entières pour les utiliser lors de la validation, afin de limiter les effets de biais liés aux caractéristiques des appelants et appelés.

De plus, dans l'objectif de s'assurer d'avoir un jeu de validation distinct du jeu d'entraînement, nous avons éliminé à la fois du jeu d'entraînement et de validation tout groupe ayant au moins une famille dans chaque jeu.

4.6 Conception de l'algorithme d'apprentissage automatique

L'algorithme utilisé est relativement simple. Basé sur un réseau de neurones, il emploie le principe des réseaux siamois. (cf. figure 7).

Trois vecteurs de caractéristiques sont fournis en entrée. Le premier vecteur est un vecteur dit de référence, le deuxième dit positif est un vecteur appartenant au même groupe de fonctions, et pour terminer un troisième vecteur dit négatif appartenant à un autre groupe de fonctions.

Le réseau qui est un réseau de type perceptron multicouche fait une projection non linéaire du vecteur d'entrée dans un espace de plus petite dimension. Cette projection est appelée vecteur de représentation.

4.7 Apprentissage

À chaque epoch, nous effectuons une permutation aléatoire pour chaque groupe de fonctions possédant au moins deux vecteurs uniques. Puis dans

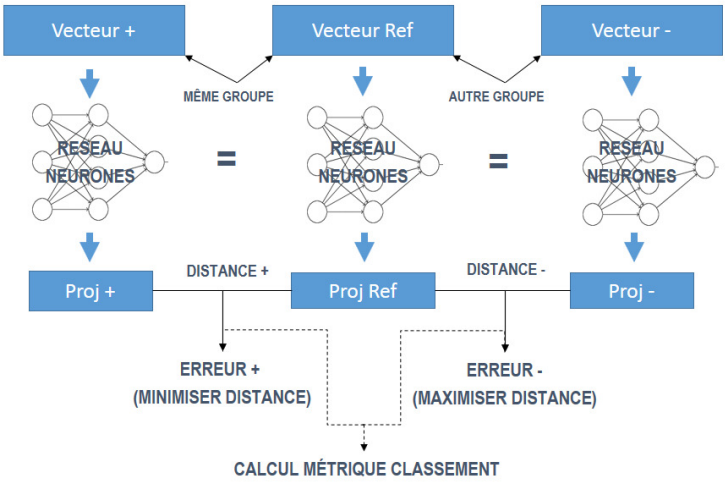


Fig. 7. Diagramme

l'ordre de la permutation nous créons une paire de vecteurs deux à deux, en rebouclant pour le dernier élement avec le premier. Un groupe possédant M vecteurs uniques aura donc M paires à chaque epoch. Par exemple si A, B, C et D sont les quatres vecteurs uniques d'un groupe de fonctions, la permutation aléatoire donnera par exemple une séquence [C, A, D, B] permettant la création des paires positives suivantes : [C, A], [A, D], [D, B] et [B, C].

Ensuite nous prenons dans les autres groupes de fonction un vecteur au hasard, ce qui permet de créer une triplette d'entraînement.

Nous utilisons ensuite la norme L_2 entre les vecteurs de représentation de référence et positif d'une part, et la norme L_2 entre les vecteurs de référence et négatifs d'autre part pour créer la fonction de coût :

$$L_+ = \|R_+ - R_-\|^2$$

$$L_- = e^{-\beta\|R_+ - R_-\|^2}$$

$$L = L_+ + L_-$$

On notera l'utilisation d'une exponentielle inverse de la norme L_{2-} afin de maximiser la distance entre vecteurs de groupes différents.

De plus, afin de mesurer au mieux la performance du modèle, nous effectuons régulièrement un classement d'un échantillon des vecteurs de validation par rapport aux autres vecteurs du jeu de validation. Une

métrique de classement est alors définie, et permet de mesurer l'écart entre le classement en cours et le classement idéal, qui serait obtenu si l'ensemble des autres vecteurs du même groupe de fonctions se retrouvaient en tête de classement. Nous calculons alors la moyenne de la somme du nombre de fonctions contenus entre la tête du classement et la position d'une fonction du groupe. Afin d'optimiser la rapidité des calculs en cours d'apprentissage, nous faisons une comparaison avec les fonctions solitaires.

Les calculs ont été effectués sur un ordinateur Xeon E5-2630 disposant de 32 Go de RAM et une Geforce GTX 1080 disposant de 8 Go de mémoire.

Nous avons entraîné le réseau sur un sous ensemble des fonctions disponibles :

Entraînement	
Nombre de vecteurs	~500 k
Nombre de vecteurs uniques familles	~100 k
Nombre de vecteurs uniques groupes	~100 k
Nombre de vecteurs uniques solitaires	~300 k

Validation	
Nombre de vecteurs	~15 k
Nombre de vecteurs uniques familles	~1 500
Nombre de vecteurs uniques groupes	~1 400
Nombre de vecteurs uniques solitaires	~1 300

Le corpus de données pèse 7.7Go (données normalisées), mais de nombreux index intermédiaires ont été ajoutés pour optimiser les accès à l'information. L'apprentissage actuel sur ce corpus réduit prend 15 minutes sans générer les classements.

Afin d'éviter le surapprentissage, plusieurs techniques de normalisation sont utilisées en parallèle : BatchNorm et Dropout.

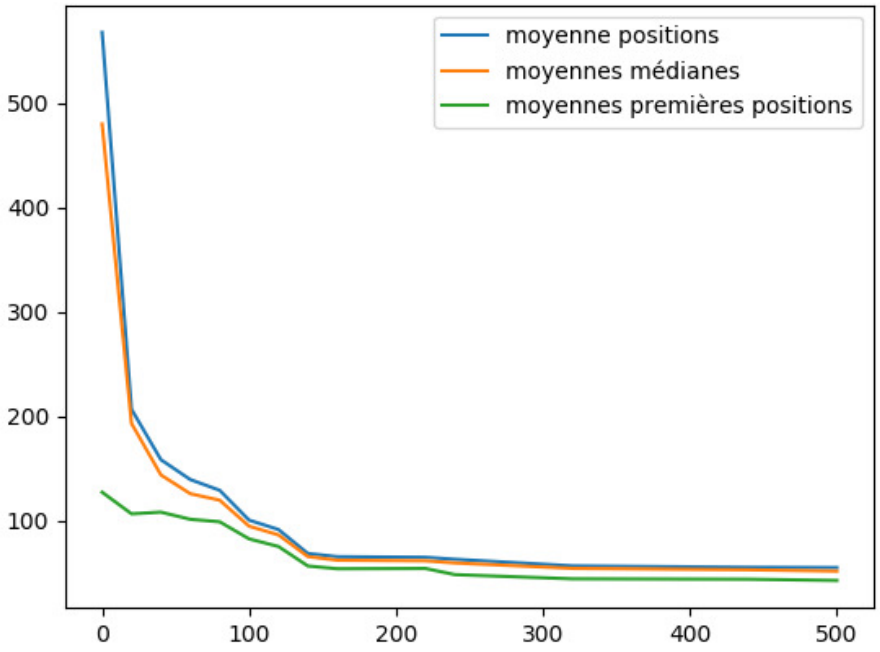
4.8 Résultats

Notre validation porte ainsi sur le binaire `nfsd.ko` : il a l'avantage d'être présent 14 fois dans le dépôt debian utilisé, et dans 3 architectures : ARM, x86 et x86_64.

Comparaison d'une fonction avec les autres versions du même binaire. Nous avons choisi 200 fonctions au hasard dans l'ensemble des

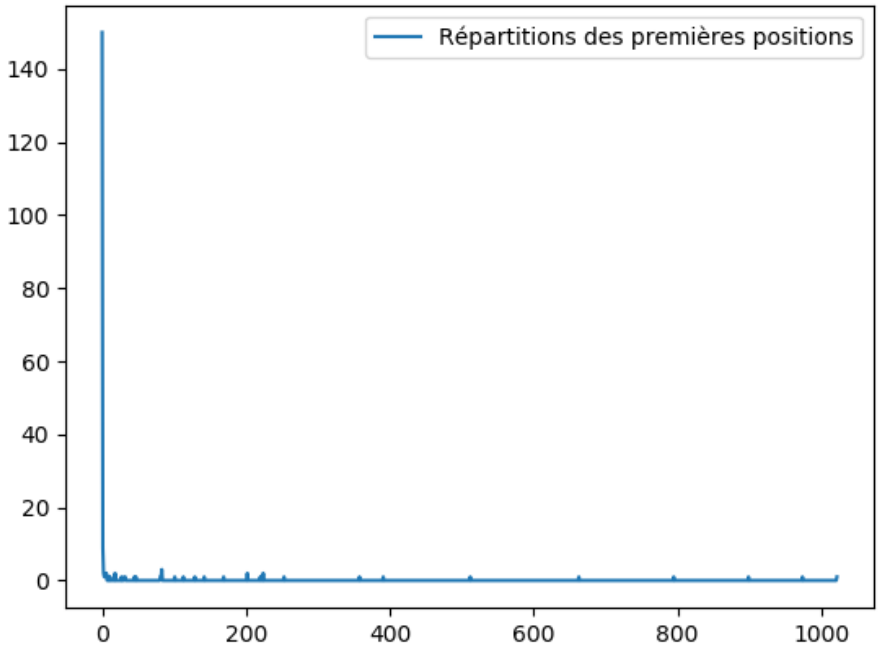
vecteurs uniques de groupes du corpus de validation. Pour chacune des fonctions, nous avons fait le classement des 1 300 fonctions solitaires et des fonctions du même groupe. Nous avons calculé plusieurs métriques, comme la métrique de classement moyenne, la médiane de classement moyenne, et la moyenne des premières positions.

Au cours de l'apprentissage nous pouvons visualiser l'évolution dans le temps de ces métriques :



En revanche, les moyennes ne permettent pas de montrer des disparités importantes dans les résultats. En effet, pour une bonne partie des fonctions testées, les résultats sont excellents (nous avons retrouvé l'ensemble des fonctions du groupe en tête de classement), alors que pour d'autres l'algorithme n'arrive pas à établir de classement correct. Ceci peut s'expliquer de part la nature non contrôlée des fonctions prises au hasard : il est ainsi tout à fait possible de choisir dans un groupe de fonctions une fonction qui a été réécrite entièrement et qui est donc très éloignée du groupe. Avec l'entraînement et l'enrichissement des corpus, on peut espérer améliorer l'algorithme y compris dans ces conditions.

En exemple, voici la répartition des rangs du premier vecteur correct correspondant à la fonction étudiée, parmi les 200 fonctions testées.



Nous pouvons remarquer que sur les 200 fonctions, nous retrouvons en première position une autre fonction du même groupe dans 150 cas. Dans le pire des cas, la première position est de l'ordre de 1 022 sur 1 300. Ce résultat est bien entendu perfectible, mais déjà particulièrement satisfaisant : l'algorithme arrive à identifier correctement la fonction la plus proche dans le corpus de validation parmis 1 300 fonctions dans 75 % des cas, et dans X % des cas la bonne fonction se trouve dans les Y premières positions.

4.9 Exploitation des résultats de l'apprentissage automatique

L'exploitation se fait de deux manières :

- Manuellement : on choisit le fichier à analyser, et on le fait correspondre avec l'ensemble des fonctions utilisées pour l'apprentissage. On trouve ainsi facilement le ou les fichiers binaires les plus proches, et pour chaque fonction, la fonction la plus proche dans la base d'apprentissage
- Avec YaDiff : une fois que l'on a analysé deux fichiers (deux versions différentes d'un même binaire par exemple), on génère un fichier de correspondances qui est alors fourni à YaDiff. Ces correspondances servent alors de base avant les autres algorithmes de propagation (comme l'algorithme de signatures).

4.10 Travaux restants

Nous allons généraliser l'approche sur un plus grand ensemble de données afin d'améliorer les résultats pour des classements globaux. Nous devons exploiter les résultats actuels afin de mieux comprendre et identifier les caractéristiques les plus pertinentes pour la classification de fonctions. Comme expliqué dans la section 4.8, on a constaté que le réseau de neurones se base beaucoup sur les appelants et appelés d'une fonction. Cela nous incite à approfondir les informations fournies de ce côté là (par exemple en donnant les moyennes des appelants des appelants, des appelants des appelés, etc.), mais également à combler les manques en fournissant plus d'éléments décrivant les paramètres internes de la fonction. Dans le cas contraire, il faut aussi s'assurer qu'il s'agit en effet d'un problème de réécriture de fonction, ou bien trouver de nouvelles caractéristiques mal prises en compte. Sur ce sujet, nous souhaiterions par exemple utiliser un élément pour l'instant ignoré par l'algorithme : les chaînes de caractères référencées par une fonction. Ces chaînes sont aujourd'hui très bien utilisées par les algorithmes utilisant les références croisées (on calcule un condensat, ce qui crée leur signature, et elles rentrent dans le lot des « objets » à faire correspondre) et l'expérience montre qu'elles augmentent de manière très importante la qualité des résultats (nombre de correspondances, taux d'erreur). Malheureusement, si cette étape est prévue, elle demeure compliquée notamment parce que le nombre et les tailles des chaînes est variable, et, comme expliqué plus haut, on gagne énormément à être capable de constituer des vecteurs de taille fixe (mais des méthodes existent, et nous prévoyons de les explorer).

5 Conclusion

À travers cet article nous apportons deux contributions distinctes : dans une première partie, un outil « tout-en-un » qui, en utilisant une succession d'algorithmes et le fruit de travaux précédents, permet de propager des informations entre deux bases IDA. Cet outil est efficace et utilisable dès aujourd'hui. Ensuite, dans une deuxième partie, nous avons présenté une méthode basée sur l'apprentissage automatique, utilisant un réseau de neurones, qui permet d'améliorer encore les résultats obtenus par la première méthode, en particulier dans des cas où, à l'heure actuelle, aucun algorithme ne montre une efficacité notable : le multi-architecture. L'état actuel de nos travaux nous permet de conclure que cette méthode est pertinente et efficace car les résultats obtenus sont déjà très bons alors que de nombreuses pistes sont encore disponibles pour l'améliorer. Elle

est déjà utilisable en l'état, et sera rapidement intégrée à l'intérieur du processus de la version « legacy » de YaDiff.

Références

1. Nicolas Economou. Turbodiff is a binary diffing tool developed as an IDA plugin. *https ://www.coresecurity.com/corelabs-research/open-source-tools/turbodiff*, 2006.
2. Thomas Dullien et Rolf Rolles. Graph-based comparison of Executable Objects. *SSTIC*, 2005.
3. Halvar Flake. Structural Comparison of Executable Objects. *DIMVA*, 2004.
4. Jennifer Widom Glen Jeh. SimRank : A Measure of Structural-Context Similarity. *SIGKDD*, 2001.
5. Matthieu Kaczmarek et Jean-Yves Marion Guillaume Bonfante. Control Flow Graphs as Malware Signatures. *WTCV*, 2007.
6. Mourad Debbabi He Huang, Amr M. Youssef. BinSequence : Fast, Accurate and Scalable Binary Code Reuse Detection. *concordia*, 2017.
7. Matt Jeongwook. ExploitSpotting : Locating VulnerabilitiesOut Of Vendor Patches Automatically. *Black Hat*, 2010.
8. Yufei Jiang Jiang Ming, Dongpeng Xu and Dinghao Wu. BinSim : Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking. *USENIX Security Symposium*, 2017.
9. Joxean Koret. Diaphora, a program diffing plugin for IDA Pro. *SyScan*, 2015.
10. pancake. radare. *LaCon*, 2008.
11. Hex Rays. IDA : the Interactive Disassembler. *https ://www.hex-rays.com/products/ida/index.shtml*.
12. Tenable Network Security. PatchDiff2 - High Performance Patch Analysis. *www.tenable.com/blog*, 2008.
13. Georg Wicherski. peHash : A Novel Approach to Fast Malware Clustering. *https ://www.coresecurity.com/corelabs-research/open-source-tools/turbodiff*, LEET.
14. xorpd. FCatalog : the functions catalog. *https ://www.xorpd.net/pages/fcatalog.html*, 2015.