Graphes de dépendances : Petit Poucet style

Mathieu Blanc, Fabrice Desclaux, Caroline Leman, Camille Mougey, Aymeric Vincent
mathieu.blanc[at]cea.fr
fabrice.desclaux[at]cea.fr
caroline.leman[at]cea.fr
camille.mougey[at]cea.fr
aymeric.vincent[at]cea.fr

CEA

Résumé. L'analyse manuelle de binaire est relativement longue, pénible et fastidieuse (ce n'est pas un métier facile). Elle lève souvent les mêmes questions : Combien d'arguments a cette fonction ? Quels sont leurs types ? D'où proviennent-ils ? Qu'est-ce qu'on mange à la cantine à midi ? Certaines de ces questions sont des problèmes très complexes qui dépendent d'une multitude de petits sous-problèmes. Cet article propose une tentative de réponse à une question simple précédemment posée : D'où provient la valeur d'une variable ?

1 Introduction

L'analyse de flot de données, *dataflow analysis*, consiste à suivre le cheminement des différentes données au fil du programme. Dans le cas d'un programme en assembleur, les données sont constituées par les registres et la mémoire. Pour analyser le flot de données, il y a deux sens possibles :

- en remontant aux sources de la variable, qui est une analyse de dépendance,
- en descendant dans les variables impactées par une variable, qui correspond à une analyse d'influence [12]. Cette analyse est également appelée analyse de teinte.

Dans cet article, nous nous concentrerons sur l'analyse des dépendances.

1.1 État de l'art

L'analyse appelée *slicing* est une application de l'analyse de dépendance. Cette technique est utilisée intuitivement par les programmeurs lorsqu'ils cherchent les bogues dans un programme informatique, comme l'a montré

M. Weiser [14]. En effet, pour savoir quel chemin d'exécution mène à une erreur, le programmeur filtre les instructions pour ne garder que celles qui ont un impact sur l'instruction posant problème.

Dans le cas concret d'une analyse statique sous *IDA*[1], le baroudeur de binaire utilise une technique redoutable : quand il veut connaître la provenance d'un registre, il le sélectionne et toutes ses occurrences sont surlignées en jaune. Un rapide coup d'oeil de l'expert permet alors de *remonter* à sa dernière affectation et d'identifier sa provenance. L'analyste peut alors réitérer le processus jusqu'à trouver la source désirée. Il remonte alors le flot de donnée pour trouver les sources d'une variable, tel le *Petit Poucet* retrouvant son chemin en observant les cailloux précédemment déposés.

Ici, on voit que le petit plus de l'outil est purement lexical : c'est le surlignement des chaînes identiques. Merci IDA !

Fig. 1. Le petit poucet semant ses cailloux

L'analyse de dépendance peut en quelque sorte être vue comme le problème inverse de l'analyse de teinte [10]. Cette dernière part d'une variable et permet de retrouver toutes les variables qui découlent de celle-ci. Si les analyses sont faites de manière globale, c'est-à-dire que

[1] Hex-Rays IDA : `https://www.hex-rays.com/products/ida/`.

toutes les dépendances de toutes les instructions sont analysées, alors le passage du *slicing* au *tainting* consiste à simplement inverser le lien de dépendance entre les variables [8]. En revanche, si les informations découlent d'une trace d'exécution du binaire, seules les instructions d'un chemin d'exécution précis sont étudiées et le tainting n'est plus le simple réciproque du slicing.

L'analyse de dépendance a de multiples cas d'application : l'optimisation de code [4, 9], désobscurcissement statique de code, analyse de *malware*, aide à la recherche de vulnérabilités... Dans la suite de l'article, nous nous intéresserons au problème de dépendance de variables dans le cas de l'analyse statique.

1.2 Langage intermédiaire

Pour réaliser leurs analyses, la plupart des compilateurs travaillent sur un langage intermédiaire. Les analyses sont indépendantes de l'assembleur cible et du code source. De la même façon notre analyse de dépendance se fait sur un langage intermédiaire. Le binaire cible sera donc dans un premier temps traduit depuis l'assembleur natif dans ce langage. Notre implémentation sera faite en utilisant le langage de Miasm [2], mais pourrait tout à fait être implémentée dans un autre langage tel que *REIL* [3]. Dans cet article, nous ne donnerons pas les détails qui permettent de traduire le code natif en langage intermédiaire, notamment la modélisation des effets de bord lors de l'appel de sous fonctions. De plus, nous nous limiterons à une analyse intraprocédurale.

2 Le Graal, vu du client

Une question que l'on peut se poser lors de l'analyse d'un programme est la suivante : dans un graphe de flot de contrôle, constitué de blocs de base [2], quelles sont les lignes assembleur qui participent à la valeur d'une variable à un point donné du programme ? Cette question nécessite quelques éclaircissements :

- une variable est ici soit un registre, soit une case mémoire,
- une ligne qui participe au calcul est une ligne assembleur qui modifie une variable qui servira *explicitement* à la génération de la variable visée.

[2] Dans la suite, on utilisera le terme *basic block*.

Le terme *explicitement* sera détaillé en section 5.3 et s'appuie sur une idée de Jean-Philippe Luyten et de Colas Le Guernic.

Cette question est soulevée dans de nombreux travaux sur l'optimisation de code, comme nous l'avons vu dans l'état de l'art. Pour illustrer les limites des algorithmes existants, nous allons présenter des cas concrets d'utilisations et nous détaillerons les résultats que nous souhaitons obtenir. Effectivement, avoir la liste des instructions qui participent à la valeur d'une variable est une chose, mais pouvoir exploiter ce résultat en est une autre.

Ces exemples sont décrits dans un *pseudo langage intermédiaire*, mais correspondent à des cas concrets en assembleur. Ceci nous permettra dans un deuxième temps de définir les solutions que devra retourner l'algorithme (une sorte de cahier des charges). Dans un troisième temps, nous introduirons une comparaison avec quelques algorithmes déjà existants. Enfin, nous nous attarderons sur des problèmes particuliers, notamment dans le cas de l'analyse de cases mémoire et de dépendances liées au graphe de flot d'exécution.

2.1 Notation

Dans la suite, on définit une variable à un point donné du programme par le terme de *DependencyNode* (ou *DepNode*).

Définition 1 (DependencyNode). *Un* DependencyNode *représente une variable à un point donné du programme. On le représente par le triplet suivant :*

label *l'adresse du* basic block *de notre variable,*
id *le nom de notre variable,*
index *le numéro de la ligne dans le* basic block*.*

L'algorithme s'exécute sur le graphe de flot de contrôle composé de *basic blocks*. Ceux-ci seront par exemple extraits d'une fonction désassemblée. Dans les graphes d'exemples suivants, les *basic block*s sont composés d'un label et d'une liste d'affectations. Le langage intermédiaire utilisé ici est fictif, mais l'algorithme reste valable pour des langages comme celui de Miasm, REIL, ...

Dans la suite, on désignera un nœud du graphe qui n'a pas de prédécesseur par le terme *tête* de graphe.

Définition 2 (*DependencyState*). *Le* DependencyState *rassemble les informations rencontrées dans l'étude d'un chemin d'exécution à partir*

d'un critère $C =< bloc_0, index_0, variables >$. *Il est composé des champs suivant :*

bloc *bloc de base de l'état représenté ;*

depgraph *un graphe associant les* DependencyNode*s qui dépendent les uns des autres sur un des chemins d'exécution entre* $bloc_0$ *et le bloc représenté,*

pendings *un dictionnaire qui relie chacune des variables dont on cherche la définition à l'ensemble des instructions, représentées par des* DependencyNode*s, qui en dépendent.*

2.2 Exemples simples

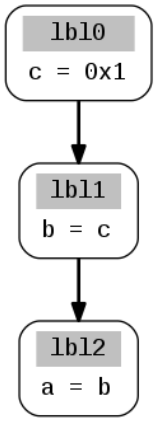

Fig. 2. Suite simple

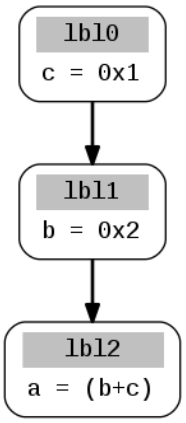

Fig. 3. Dépendance multiple

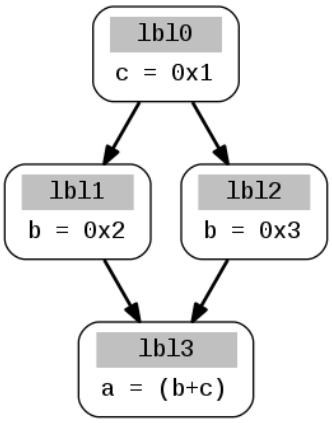

Fig. 4. Condition aboutissant à des solutions multiples

Cas triviaux Dans l'exemple de la figure 2, nous désirons obtenir les dépendances de la variable *a* dans le *basic block lbl2*, juste après son affectation depuis *b*. On représente cette cible avec la notation *DepNode(lbl2, a, 1)*, c'est-à-dire la variable du *basic block lbl2* qui a pour nom *a* à la ligne 1. Les numéros de lignes commencent à 0 et sont les numéros de lignes dans les *basic block*s en langage intermédiaire. On peut voir que *a* dépend de *b*, qui dépend de la variable *c*, qui quant à elle dépend de la constante `0x1`.

Un point intéressant est que dans cet exemple, à la fin du *basic block lbl2*, *a* vaut forcément `0x1`. La solution de l'algorithme devra pouvoir être utilisée pour obtenir cette information.

Dans la figure 3, on voit que *DepNode(lbl2, a, 1)* dépend de *DepNode(lbl2, b, 0)* et de *DepNode(lbl2, c, 0)*. *DepNode(lbl2, b, 0)* dépend de *DepNode(lbl1,* `0x2`*, 0)*. *DepNode(lbl2, c, 0)* dépend de *DepNode(lbl0,* `0x1`*, 0)*. On aura donc ici une seule solution, qui nous informe que *DepNode(lbl2, a, 1)* dépend du couple *DepNode(lbl1,* `0x2`*, 0)*, *DepNode(lbl0,* `0x1`*, 0)*. De même on voudra pouvoir exploiter le résultat pour conclure ici que *a* vaut toujours 3.

Solutions multiples Dans la figure 4, le problème se complique. On voit que *DepNode(lbl3, a, 1)* dépend de *DepNode(lbl3, b, 0)* et de *DepNode(lbl3, c, 0)*. Le souci est que le *basic block lbl3* a deux parents. Si on considère l'algorithme de base qui renvoie les lignes qui participent à la valeur finale d'une variable cible, la sortie sera un ensemble contenant toutes les lignes du graphe. Néanmoins, ici le code peut provenir soit de la branche de droite, soit de la branche de gauche.

Nous avons fait le choix de renvoyer dans ce cas deux solutions indépendantes. Une solution remonte le graphe de flot d'exécution en utilisant la branche de gauche, ce qui nous permettra de trouver une première solution dans laquelle *DepNode(lbl3, a, 1)* dépend de *DepNode(lbl1,* `0x2`*, 0)* et *DepNode(lbl0,* `0x1`*, 0)*. Une autre solution est renvoyée en remontant la branche de droite, dans laquelle *DepNode(lbl3, a, 1)* dépend de *DepNode(lbl2,* `0x3`*, 0)* et *DepNode(lbl0,* `0x1`*, 0)*.

Un schéma étant plus parlant qu'une longue explication, la suite décrira les solutions sous forme de graphes de dépendances. Le graphe d'une solution est défini par :

- ses nœuds : les *DependencyNode*,
- ses arcs définis comme suit : un arc existe entre deux nœuds si la variable définie dans le nœud source est impliquée dans le calcul de la variable présente dans le nœud de destination.

L'analyse du graphe de flot de contôle de la figure 4 devra retourner les solutions représentées dans les figures 5 et 6. On peut noter que leur exploitation permet de déduire dans chaque cas la valeur finale de *a*.

Exploitation Dans les cas simples que nous avons vus jusqu'ici, nous aimerions pouvoir récupérer la valeur finale de la variable cible. Pour cela, l'algorithme doit noter l'historique du parcours des *DependencyNode*s. Dans un deuxième temps, on peut réaliser une émulation symbolique des *DependencyNode*s renvoyés. Dans ces cas simples, on peut noter que les solutions de l'algorithme sont représentables sous forme d'arbre de dépendances.

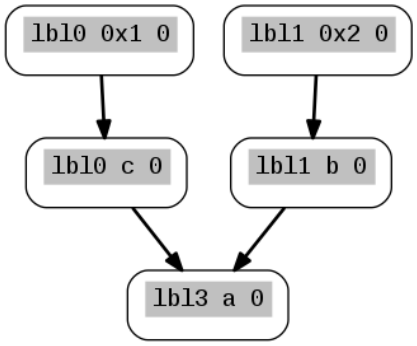

Fig. 5. Branche de gauche

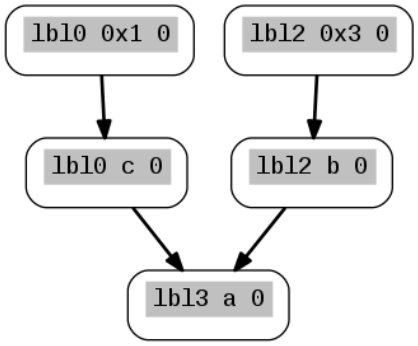

Fig. 6. Branche de droite

Boucles Dans les cas suivants, l'affectation à la variable *pc* permet de définir le *basic block* suivant à exécuter.

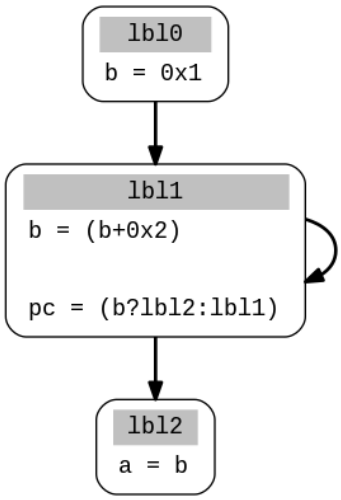

Fig. 7. Boucle

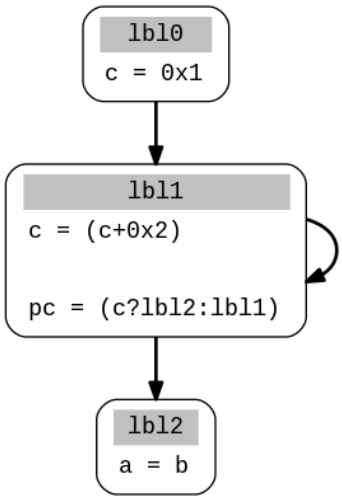

Fig. 8. Boucle sans impact

La figure 7 représente un graphe de flot de contrôle où une variable impliquée dans la dépendance de `a` est modifiée dans une boucle. La méthode que nous avons retenue ne renvoie que des solutions qui ont des graphes de dépendances différents. Dans l'exemple de la figure 7, l'algorithme renverra deux solutions : la première donnera les dépendances dans le cas où le *basic block lbl1* n'est exécuté qu'une seule fois, et la deuxième dans le cas où ce *basic block* est exécuté au moins deux fois.

La première solution (figure 9) est obtenue si on extrait le graphe de dépendances sans boucler. La deuxième solution (figure 10) représente les dépendances si on exécute au moins deux fois le *basic block lbl1*. Une

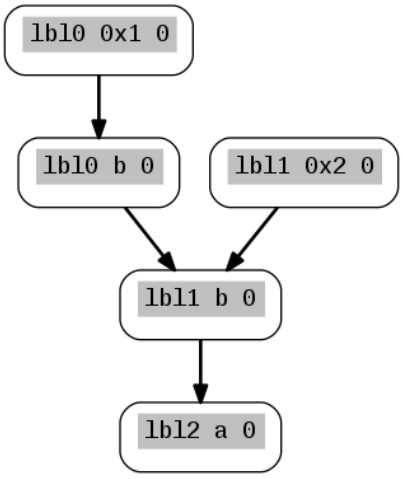

Fig. 9. Solution 1

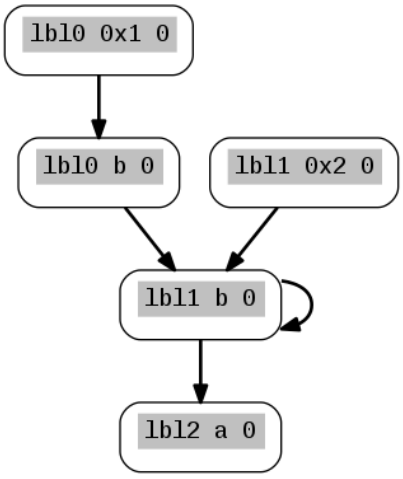

Fig. 10. Solution 2

boucle apparaît également dans les dépendances, sur la variable qui est modifiée dans le corps de la boucle du flot d'exécution.

La figure 8 est très similaire à la figure 7. On voit que *DepNode(lbl2, a, 1)* dépend de *DepNode(lbl2, b, 0)* dans un premier temps. En outre, on peut tirer la dépendance de *DepNode(lbl2, b, 0)* jusqu'à la tête du graphe de flot d'exécution (que l'on prenne une ou plusieurs fois la boucle) sans que la solution ne soit impactée. La solution sera ici unique et on aura *DepNode(lbl2, a, 1)* qui dépend uniquement de *DepNode(lbl0, b, 0)*. Ici on obtient un seul graphe solution (figure 11).

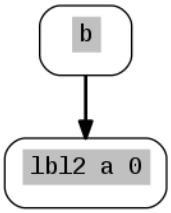

Fig. 11.

Nous allons maintenant présenter des cas problématiques qui nous ont poussé à adapter les algorithmes existants.

2.3 Limites des solutions existantes

L'analyse de dépendances est un domaine qui a déjà été beaucoup étudié et des études des solutions existantes sont disponibles [12].

On pourra notamment citer des solutions se basant sur le *Program Dependence Graph* [4], permettant par exemple de détecter des fuites d'informations [5]. Ces analyses ont aussi été étendues à de l'inter-procédural [6]. D'autres solutions se basent sur de l'exécution symbolique, avec diverses méthodes d'améliorations pour éviter l'explosion combinatoire [1, 7].

Enfin, certaines solutions extraient l'ensemble des dépendances sans tenir compte des chemins d'exécutions [11].

Plusieurs solutions sur étagères (celles du marchand de cailloux) sont donc disponibles. Elles peuvent globalement être regroupées en deux catégories :

- la détection de dépendance entièrement statique sans tenir compte des chemins d'exécutions ;
- la détection de dépendance en résolvant indépendamment les différents chemins avec leurs contraintes associées (que l'on pourrait considérer presque dynamique).

Pour notre *Petit Poucet*, cela revient à choisir entre une solution contenant trop peu d'information (sac de sable) et une solution potentiellement très lourde à utiliser (sac de menhirs).

Spoiler : l'algorithme décrit dans cet article est à mi-chemin entre ces deux solutions (sac de *pocket*-cailloux). Informellement, il se veut capable de distinguer les chemins d'exécutions, de rassembler les chemins «équivalents en dépendance» et de dérouler les boucles seulement le nombre de fois nécessaire (borné par le nombre d'instructions dans les *basic blocks* du corps de la boucle).

Pour illustrer ces différences, nous comparons la sortie de notre algorithme avec trois algorithmes implémentés dans d'autres outils :

- le *BackwardSlice* et le *PathGroup* d'Angr[3] ;
- le *Backtrace* de Metasm[4].

Ces cas d'étude reflètent l'usage de ces outils par les auteurs de cet article et ne sauraient en représenter une utilisation optimale. Malgré notre attention, il est possible que des erreurs dans leur emploi aient été commises.

Cas d'étude L'exemple qui nous servira de support pour les comparaisons est représenté dans le listing 1.

[3] Angr : `http://angr.github.io`
[4] Metasm : `http://metasm.cr0.org/`

```
1  entrypoint:
2      mov         ecx, 1
3      mov         edx, 2
4      jmp         loop
5
6  loop:
7      inc         edi
8      mov         ebx, ecx
9      mov         ecx, edx
10     cmp         edi, 10h
11     jnz         loop
12
13 end:
14     mov         eax, ebx
15     ret
```

Listing 1. Programme de comparaison

Cet exemple est composé d'un bloc d'initalisation (`entrypoint`), d'un bloc qui boucle sur lui-même (`loop`) et d'un bloc de fin (`end`).

On remarque que l'arrêt de la boucle est conditionné par `edi`, qui n'est jamais initialisé. Le nombre de passages dans la boucle dépend donc directement de la valeur initiale, avant `entrypoint`, de `edi`.

Nous sommes intéressés par la valeur finale de `eax` (à la fin du bloc `end`). Ce registre peut prendre seulement deux valeurs :

- `eax = 1` : dans le cas où la boucle n'est parcourue qu'une seule fois (qu'elle ne boucle pas, donc). Cette valeur s'obtient seulement avec la condition initiale `edi = 0xf` ;
- `eax = 2` : dans le cas où au moins un tour de boucle est effectué, on aura `ecx = edx = 2` (ligne 9, ligne 3). Cette valeur s'obtient avec la condition initiale `edi != 0xf`.

BackwardSlice L'algorithme *BackwardSlice*, implémenté dans Angr (`angr.analyses.backward_slice::BackwardSlice`), appartient à la catégorie des algorithmes statiques sans distinction de chemins d'exécution.

Nous lançons l'algorithme avec pour entrée le registre `eax` ligne 15. Le code utilisé est disponible en Annexe A.

Le résultat de l'algorithme est obtenu dans le langage intermédiaire d'Angr[5] et est rendu grâce au moteur de graphe de Miasm (créant ainsi un caillou *à la Frankenstein*) sur la figure 13. En ne conservant que les lignes assembleur concernées, on obtient le *listing* de la figure 12.

Cet algorithme a les avantages suivants :

- entièrement statique ;

[5] `https://github.com/angr/angr-doc/blob/master/docs/ir.md`

```
1  entrypoint:
2      mov         ecx, 1
3      mov         edx, 2
4      ; jmp         loop
5
6  loop:
7      inc         edi
8      mov         ebx, ecx
9      mov         ecx, edx
10     cmp         edi, 10h
11     jnz         loop
12
13 end:
14     mov         eax, ebx
15     ; ret
```

Fig. 12. Report de la figure 13 sur l'assembleur

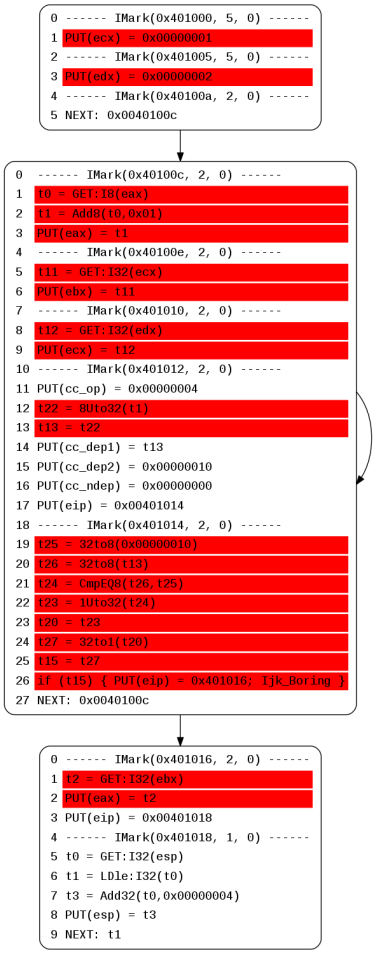

Fig. 13. BackwardSlice en IR Angr (en rouge les lignes appartenant au *slice*)

- très rapide (de l'ordre de $\mathcal{O}(n+e)$, où n est le nombre de noeud et e le nombre d'arrête du CFG) ;
- toutes les lignes nécessaires ont effectivement été trouvées.

Et les inconvénients :

- impossibilité de distinguer les chaînes de dépendances pour les différentes valeurs du `eax` final ;
- impossibilité de connaître les éléments utilisés conjointement. Par exemple, les lignes 2 et 3 ne servent jamais pour la même solution ;
- pas de valeur finale pour `eax`.

Cet algorithme est donc très pratique pour isoler rapidement les lignes *potentiellement* utilisées dans le calcul d'une valeur particulière.

En revanche, il ne peut pas être utilisé directement pour, par exemple, récupérer les couples de valeurs des registres à la fin de la fonction (`eax=1, ebx=1` ou `eax=2, ebx=2`, mais pas `eax=1, ebx=2` ou `eax=2, ebx=1`).

Ces registres pourraient être utilisés près d'une vulnérabilité potentiellement conditionnée par leurs valeurs (condition `eax == ebx`), ou en arguments d'appel à une fonction.

PathGroup L'algorithme *PathGroup*, implémenté dans Angr (`angr.path_group::PathGroup`), appartient à la catégorie des algorithmes dits *path-sensitive*.

Cet algorithme va trouver un ensemble de chemins d'exécution *réalisables*, entre un bloc de départ et un état final (par exemple, l'arrivée sur un bloc donné ou sur un accès mémoire contrôlé).

Intuitivement, il va utiliser de l'exécution symbolique pour parcourir les blocs du CFG au fur et à mesure, tout en vérifiant leur faisabilité (la possibilité de les atteindre en choisissant des valeurs initiales données). Cette méthode est aussi appelée *Dynamic Symbolic Execution*[6].

Cet algorithme va donc :

1. en atteignant une première fois la ligne 11, trouver une valeur de `edi` initiale (0xf) permettant d'atteindre `end` et une autre ($\notin \{0xf\}$) permettant de continuer la boucle ;
2. en atteignant une seconde fois la ligne 11, trouver une valeur de `edi` initiale (0xe) permettant d'atteindre `end` et une autre ($\notin \{0xf, 0xe\}$) permettant de continuer la boucle ;
3. ... et ainsi de suite, pour les 2^{32} possibilités de valeur initiale de `edi`.

Une version naïve de cet algorithme ne finirait donc pas en un temps raisonnable (comprendre «humainement utilisable»). De plus, l'ordre dans lequel les chemins possibles vont être traités est ici arbitraire ; les chemins finissant sur `ret` pourraient n'être traités qu'en dernier, laissant l'utilisateur sans aucun résultat en un temps raisonnable.

Pour éviter ces écueils, ces types d'algorithme utilisent des heuristiques ou font des concessions sur la précision de leur résultat.

Dans le cas du *PathGroup* (voir code en Annexe B), cela se traduit par seulement un résultat. En effet, un seul chemin a été trouvé entre le bloc initial et le bloc `end`, avec pour état final (en rentrant dans le bloc `end`, donc nous nous intéressons à `ebx`) :

- `edi` = 0xf ;

[6] Triton - *Dynamic Symbolic Execution engine* : `http://triton.quarkslab.com/`

– `ebx` = 1.

Le cas où `eax` final vaut 2 (`ebx` intermédiaire vaut 2) est donc ignoré.

Ce type d'algorithme peut être très efficace pour de l'analyse de DRM ("trouve un chemin et les entrées utilisateur associées pour atteindre le bloc de succès", voir les exemples Angr[7]) ou pour de la recherche de vulnérabilités (l'équipe d'Angr, «ShellPhish», a été qualifiée pour la finale du *DARPA Cyber Grand Challenge*[8]).

Cependant, c'est une analyse dynamique, qui peut être trop lourde pour certains besoins. Si un sous-graphe (par exemple, une boucle de décompression) n'introduit pas de dépendance au regard des variables traquées, il devra quand même être parcouru et résolu avant de pouvoir atteindre le bloc désiré. Dans le cas d'étude présent, même si la boucle n'introduit plus de nouvelle dépendance après le second tour, elle devra tout de même être analysée en considérant son exécution n fois.

Finalement, l'utilisateur obtiendra un résultat potentiellement plus précis, mais seulement *si l'algorithme termine*. Ce type d'analyse est donc trop lourde pour les besoins du Petit Poucet (sac de menhirs).

Metasm L'algorithme *Backtrace* de Metasm (`metasm/disassemble::backtrace`) se rapproche d'une analyse de dépendance en intégrant la notion de chemin dans les solutions fournies. Pour palier le problème des boucles, l'algorithme s'interdit de repasser deux fois dans un même bloc. Sa sortie donne, en plus des résultats, l'information qu'il n'a pas suivi certaines solutions liées à cette limitation.

Dans notre cas (code en Annexe C), l'algorithme retourne une seule solution : `eax` vaut 1.

Le fait de ne pas parcourir plus d'une fois les boucles lui permet de traverser rapidement les parties inutiles (n'introduisant pas de nouvelle dépendance) du CFG, le tout en statique. De plus, l'algorithme est capable d'évaluer la valeur finale de la variable traquée. Enfin, l'algorithme indique qu'il passe potentiellement à côté de certaines solutions lorsque c'est le cas.

Cette analyse, tout comme l'algorithme décrit dans cet article, se trouve donc à mi-chemin entre les algorithmes présentés plus haut : il est suffisamment rapide et conserve assez de précision pour être utilisable dans nos cas d'étude.

[7] Exemples Angr : `https://github.com/angr/angr-doc/blob/master/docs/examples.md`

[8] DARPA Cyber Grand Challenge : http ://www.cybergrandchallenge.com/

Cependant, le fait d'ignorer les dépendances introduites par de multiples passages dans un même bloc peut entraîner une perte non négligeable de précision des solutions (même si l'utilisateur en est informé). Les auteurs de cet article considèrent qu'il est possible de réduire ces pertes tout en conservant des temps de calcul acceptables, et ce grâce à l'algorithme du *DepGraph*.

DepGraph Sur l'exemple précédent, l'algorithme du *DepGraph* (utilisé dans l'exemple `examples/symbol_exec/depgraph.py` de Miasm) retourne deux solutions :

- `eax = 1`, en conservant les lignes 2, 8 et 14 ;
- `eax = 2`, en conservant les lignes 3, 9, 8 et 14 (dans l'ordre de leur dépendance).

Ainsi, l'algorithme est capable de calculer statiquement les dépendances en différenciant deux chemins d'exécution introduisant des dépendances distinctes. De plus, il est possible d'obtenir la valeur finale de `eax`. Dans le cas où la valeur finale serait indéterminée, par exemple la valeur de `eax` avant d'exécuter la ligne 14, l'algorithme retournera une valeur possible tout en indiquant qu'il y a potentiellement d'autres solutions.

La section 5.4 donne un aperçu des temps de traitement de l'algorithme, qui sont considérés par les auteurs comme étant raisonnables ("humainement utilisable").

Nous allons maintenant détailler le fonctionnement de l'algorithme du *DepGraph*.

3 Algorithme

Le but de l'algorithme est d'être suffisamment précis pour distinguer des chemins d'exécutions pertinents, tout en évitant l'explosion combinatoire d'un parcours exhaustif des chemins d'exécution.

Pour représenter l'état des dépendances au fur et à mesure du déroulement de l'algorithme, nous avons introduit un objet *DependencyState*, décrit par la définition 2 p. 409.

L'idée est de suivre les dépendances en partant d'un *basic block* donné. Si le *basic block* a plusieurs parents, les dépendances sont suivies séparément dans ces derniers. Si on arrive à un parent commun avec les mêmes *DependencyState*s, on n'en suivra plus qu'un seul. Plus généralement, si on atteint un *basic block* dont le *DependencyState* a déjà été traité, ce chemin d'analyse s'arrête. Enfin, si on atteint une tête du graphe ou

un *DependencyState* dans lequel toutes les dépendances ont été résolues, une solution construite à partir du *DependencyState* est renvoyée. Ce mécanisme déroule les boucles tant qu'une nouvelle itération introduit de nouvelles dépendances. De même, les sous graphes qui n'introduisent pas de nouvelles dépendances sont ignorés.

L'algorithme de calcul des dépendances (voir algorithme 1) prend en entrée un graphe de flot de contrôle (CFG), un *basic block*, `blocbasique`, un numéro de ligne, `index`, et un ou plusieurs noms de variable, `variables`, dont nous souhaitons calculer les dépendances à partir du bloc `blocbasique` à l'instruction `ligne`. Ce paramètre est similaire au critère de tranche tel que défini par Weiser [13].

Le *DependencyState* représente pour un bloc donné les dépendances rencontrées sur un chemin d'exécution entre le bloc et l'instruction du critère, afin d'identifier les blocs qui, sur différents chemins d'exécution, introduisent exactement les mêmes dépendances.

L'algorithme utilise une *work list* et procède par itérations sur les blocs en remontant les prédécesseurs. Pour chaque bloc, un nouveau *DependencyState* est créé à partir du *DependancyState* du successeur de ce bloc dans le CFG.

Les 3 étapes suivantes sont effectuées sur le nouveau *DependencyState* :

1. la propagation intra-bloc (ligne 10 de l'algorithme 1) : identification des dépendances pertinentes à l'intérieur d'un *basic block* ;
2. la reconnaissance d'un état déjà atteint (ligne 11 de l'algorithme 1) : vérifier que le *DependencyState* n'a pas déjà été rencontré ;
3. le passage aux blocs suivants (ligne 21 de l'algorithme 1) : propagation du *DependencyState* courant aux prédécesseurs dans le graphe de flot de contrôle.

Propagation intra-bloc (algorithme 2) Pour toutes les variables présentes dans le dictionnaire des dépendances *pendings* du *DependencyState*, on recherche dans le *basic block* concerné par le *DependencyState* les lignes qui affectent ces variables. Cette recherche est faite en parcourant dans l'ordre inverse la liste des instructions du *basic block*, car on cherche la définition des variables la plus récente (on *remonte* ici les dépendances d'une variable) :

- si la ligne courante ne définit aucune variable des clés du *pendings*, on passe à la ligne précédente ;
- si la ligne définit une des variables des clés du *pendings*, *var*, alors on a trouvé une définition de laquelle dépendent tous les *DependencyNodes*

Algorithme 1 : Graphe de dépendance

Entrées : *cfg* : graphe de flot de contrôle, *blocbasique*, *index*, *variables*
Sorties : Forêt de graphes de dépendances

```
1  pending_initiaux = dictionnaireVide() ;
2  pour chaque var ∈ variables faire
3      pending_initiaux [var] = ensembleVide() ;
4  fin
5  état_initial = DState(blocbasique, graphe_vide(), pending_initiaux) ;
6  todo = ensemble(état_initial) ;
7  resultat = ensembleVide() ;
8  tant que todo faire
9      état = todo.pop() ;
10     analyse_intra_bloc(cfg, état) /* Voir algorithme 2 */
11     si état ∈ fait alors
12         passer au suivant ;
13     fin
14     fait = fait ∪ {état} ;
15     si est_vide(état.pendings) ou est_une_tête(cfg, état.bloc) alors
16         resultat.ajouter(état) ;
17         passer au suivant ;
18     fin
19     pour chaque préd ∈ prédécesseurs(cfg, état.bloc) faire
20         état_prédécesseur = DState(préd.bloc, état.depgraph, état.pendings) ;
21         todo.ajouter(état_prédécesseur) ;
22     fin
23 fin
24 retourner resultat
```

Algorithme 2 : Analyse intra-bloc

Entrées : *cfg*, état
Sorties : Modification de état par effet de bord

```
1  pour chaque instr dans l'ordre inverse d'exécution de état.bloc.instructions faire
2      si lval(instr).est_une_clé(état.pendings) alors
3          nœud_définition = DNode(état.bloc, lval(instr), instr) ;
4          état.depgraph.ajouter_nœud(nœud_définition) ;
5          nœuds_utilisateur = copier(état.pendings[lval(instr)]) ;
6          pour chaque noeud nœud_utilisation ∈ nœuds_utilisateur faire
7              état.depgraph.ajouter_arc(nœud_utilisation,nœud_définition) ;
8          fin
9          état.pendings.supprimer(lval(instr)) ;
10         pour chaque variable var ∈ READ(instr) faire
11             état.pendings[var] = état.pendings[var] ∪ {nœud_définition} ;
12         fin
13     fin
14 fin
```

liés à cette variable dans le dictionnaire *pendings*. Dans ce cas, on crée un *DependencyNode DN* représentant la variable *var* à l'instruction où elle est définie. On met ensuite à jour les différents champs du *DependencyState* :

depgraph on relie chaque noeud dépendant de *var* dans *pendings* à *DN*.

pendings on efface l'entrée correspondant à *var* (cette dépendance est maintenant résolue) puis on rajoute les nouvelles dépendances en liant dans le dictionnaire *pendings* chacune des variables utilisées pour définir *var* à *DN*. Notons qu'il est possible que ces variables soient déjà des clés de *pendings* auquel cas *DN* est ajouté dans les valeurs associées à *var* dans *pendings*.

Reconnaissance d'un état déjà atteint Après avoir terminé l'analyse d'un bloc, nous tentons de découvrir si cet état a déjà été rencontré (ligne 11 de l'algorithme 1). Il y a deux possibilités : soit le bloc étudié a déjà été traité en passant par un autre prédécesseur (bloc de condition par exemple), soit il a déjà été traité précédemment (cas d'une boucle par exemple).

Dans le premier cas, nous souhaitons obtenir deux solutions distinctes seulement si les deux chemins introduisent des dépendances différentes, c'est-à-dire qu'au moment du traitement du bloc, les graphes de dépendances *temporaires* sont différents en fonction du chemin étudié. Ainsi, pour ce bloc, les *DependencyState*s sont différents.

Dans le second cas, nous avons décidé d'appliquer le même raisonnement. Si le fait de boucler n'introduit pas de nouvelles dépendances, alors le *DependencyState* est identique. La boucle n'est déroulée que le nombre de fois minimum pour trouver toutes les dépendances qu'elle pourrait générer. Nous donnons plus de détails concernant la majoration du nombre de solutions dans la partie 4.

Passage aux prédécesseurs Le passage aux prédécesseurs dans le graphe de flot de contrôle consiste à propager les dépendances des variables déjà résolues (*depgraph* ainsi que les variables qu'il reste à résoudre *pendings*) au prédécesseur. Un nouvel état est créé pour chaque bloc prédécesseur, permettant ainsi de séparer l'analyse des différents chemins. Les analyses seront comparées lors de la rencontre du bloc de branchement à l'origine de la séparation des chemins d'exécution.

3.1 Terminaison de l'algorithme 1

On remarque que `analyse_intra_bloc` termine toujours car elle effectue un parcours des instructions d'un bloc de base.

Nous allons montrer que l'ensemble `fait` ne peut croître infiniment : en effet, il est modifié à la ligne 14, à laquelle un *nouvel* élément lui est ajouté, comme l'assure le test qui précède. Le nombre de graphes possibles dans `état.depgraph` est borné. De plus le nombre possible de labels dans `état.bloc` est aussi borné par la taille du CFG. Le cardinal de `fait` est donc aussi borné.

Si `todo` croît, alors `fait` croît, et si `fait` ne croît pas, `todo` décroît. Ainsi il n'est pas possible de boucler indéfiniment.

3.2 Exemple d'exécution de l'algorithme

La suite illustre l'algorithme en le déroulant sur l'exemple du graphe de flot de contrôle de la figure 14.

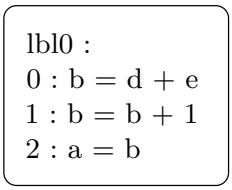

Fig. 14. Exemple à un bloc

Si on demande le suivi de *a*, le *DependencyState* initial sera d'abord composé d'un *depgraph* vide et d'un ensemble :

```
pendings:
a : None
```

A l'issu de l'analyse intra-bloc, le *DependencyState* associé au bloc `lbl0` est le suivant :

```
bloc: lbl0
depgraph:
DN(lbl0,a,2) -> DN(lbl0,b,1)
DN(lbl0,b,1) -> DN(lbl0,b,0)
pendings:
d : DN(lbl0,b,0)
e : DN(lbl0,b,0)
```

Traitement des boucles Étudions le résultat obtenu sur un programme contenant des boucles, comme celui de la figure 7 page 411. On cherche à connaître *a* à la fin du bloc *lbl2*.

L'analyse du premier *basic block* (*lbl2*) donne le *DependencyState* suivant :

```
bloc: lbl2
depgraph:
(lbl2, a,0)
pendings:
b : (lbl2, a,0)
```

Ce *DependencyState* est sauvegardé dans l'ensemble *fait*, puis il est dupliqué et propagé à son unique prédécesseur, le bloc `lbl1`.

```
bloc : lbl1
depgraph:
(lbl2,a,0) -> (lbl1,b,0)
pendings:
b : (lbl1,b,0)
```

Ce *DependencyState* n'a pas encore été rencontré, c'est pourquoi nous allons dupliquer et propager cet état aux deux parents du bloc *lbl1*, les blocs `lbl0` et `lbl1`. Les analyses de ces deux blocs donnent les *DependencyState*s suivants :

```
bloc: lbl0                  bloc: lbl1
depgraph:                   depgraph:
(lbl2,a,0) -> (lbl1,b,0)    (lbl2,a,0) -> (lbl1,b,0)
(lbl1,b,0) -> (lbl0,b,0)    (lbl1,b,0) -> (lbl1,b,0)
(lbl0,b,0) -> (lbl0,0x1,0)  pendings:
pending:                    b : (lbl1,b,0)
```

Pour le bloc `lbl0`, l'ensemble *pendings* est vide, donc les dépendances ont toutes été résolues : cet état est retourné. Concernant le bloc `lbl1`, on constate que cet état n'a pas encore été rencontré donc on propage ce *DependencyState* à ses prédécesseurs et l'analyse des dépendances donne les deux *DependencyState*s suivants :

```
bloc: lbl0
depgraph:
(lbl2,a,0) -> (lbl1,b,0)
(lbl1,b,0) -> (lbl1,b,0)
(lbl1,b,0) -> (lbl0,b,0)
(lbl0,b,0) -> (lbl0,0x1,0)
pending:
```

```
bloc: lbl1
depgraph:
(lbl2,a,0) -> (lbl1,b,0)
(lbl1,b,0) -> (lbl1,b,0)
pendings:
b : (lbl1,b,0)
```

L'état pour le bloc `lbl0` n'a pas été vu précédemment (le *depgraph* possède l'arc `(lbl1,b,0) -> (lbl1,b,0)` en plus), et *pendings* est vide, l'état est donc retourné.

En revanche, l'état correspondant au bloc `lbl1` a déjà été rencontré, un nouveau parcours n'engendrera pas de nouvelles dépendances, donc l'analyse s'arrête ici.

L'algorithme retournera alors les deux *DependencyState*s :

```
bloc: lbl0
depgraph:
(lbl2,a,0) -> (lbl1,b,0)
(lbl1,b,0) -> (lbl1,b,0)
(lbl1,b,0) -> (lbl0,b,0)
(lbl0,b,0) -> (lbl0,0x1,0)
pending:
```

```
bloc: lbl0
depgraph:
(lbl2,a,0) -> (lbl1,b,0)
(lbl1,b,0) -> (lbl0,b,0)
(lbl0,b,0) -> (lbl0,0x1,0)
pending:
```

Ces résultats représentent deux types d'exécutions différents : le deuxième, dans lequel le corps de la boucle n'est exécuté qu'une seule fois, et le premier, dans lequel la boucle peut être exécutée deux fois ou plus.

3.3 Limites

L'algorithme ne connaît pas de graphe d'appels et ne remonte donc pas les appels de fonctions : il est *intra-procédural*.

Une autre limite de l'algorithme est le suivi des cases mémoire. En effet, le suivi de variables à travers des manipulations de registres est relativement aisé car on peut déduire immédiatement d'une ligne assembleur si un registre est modifié ou non. Il n'en est pas de même si on passe par une case mémoire. Prenons le cas du code suivant :

```
1 mov [ebx], 0x6
2 mov [ecx], 0x18
3 mov eax, [ebx]
```

Listing 2. Aliasing d'adresses

Nous désirons connaître la valeur de `eax` à la ligne 3. Elle dépend de la case mémoire `[ebx]`. Cette dernière est d'ailleurs affectée à la ligne 1. On pourrait en déduire que `eax` vaut 6. Néanmoins, si on analyse la ligne 2, on voit qu'une affectation a lieu dans la case mémoire `[ecx]`. Nous avons alors plusieurs cas possibles :

- `ebx` et `ecx` ont la même valeur, et sont donc des alias sur la même case mémoire. Dans ce cas `eax` vaudra 0x18,
- les cases mémoire `[ebx]` et `[ecx]` sont parfaitement disjointes, et ici `eax` vaudra 0x6,
- les cases mémoire se recouvrent, et `eax` aura une autre valeur.

Le suivi des dépendances devient ici complexe. Une façon de le résoudre serait de faire au préalable une analyse de `ebx` et `ecx` pour savoir dans quel cas nous sommes.

Ici, le Petit Poucet a semé des bouts de pain à la place des cailloux. Des oiseaux ont mangé ou même déplacé des bouts de pain pendant leur festin. Sur le retour le Petit Poucet se retrouve avec un problème inextricable et ne sait pas comment retrouver son chemin. Il en est de même sur la recherche des dépendances de nos variables.

Fig. 15. Voilà ce qui arrive quand on ne suit pas les bonnes dépendances

Pour le moment, nous allons considérer que deux cases mémoire sont identiques si leurs adresses sont syntaxiquement les mêmes. Ceci est faux

dans le cas général, mais reste «souvent» utile. Le cas classique est le suivi de données stockées dans la pile. Ce point sera abordé dans la section 5.3.

4 Evaluation du nombre de solutions

Nous souhaitons évaluer le nombre de solutions retournées par l'algorithme 1 p. 420. Pour ce faire, nous allons formaliser les objets qui ont été introduits précédemment, ce qui nous permettra de prouver la borne que nous obtenons.

Nous montrerons que chacune des adaptations ne change pas l'ordre de grandeur du nombre de solutions.

4.1 Adaptation de l'algorithme

Dans cette partie, nous fournissons la nouvelle version de l'algorithme facilitant les preuves.

Entrées

- un graphe de flot de contrôle $\langle V, E, t, X, \mathit{lval}, \mathit{use}\rangle$ (définition 3),
- un critère, $\langle n_c, X_c\rangle$ composé d'un nœud $n_c \in V$ et d'un ensemble de variables $X_c \subseteq X$.

Le CFG, par rapport à la représentation intermédiaire de Miasm, n'a qu'une tête et ses nœuds contiennent exactement une affectation. De ce fait par exemple, le graphe abstrait les conditions de saut tout en conservant les arcs.

Définition 3 (Control Flow Graph, Graphe de flot de contrôle, CFG).

Un CFG est un graphe connexe $\langle V, E, t, X, \mathrm{lval}, \mathrm{use}\rangle$ *où* V *est un ensemble fini de* nœuds *représentant une affectation,* $E \subseteq V \times V$ *un ensemble d'arcs indiquant les successeurs d'un nœud,* $t \in V$ *est le seul nœud sans prédécesseur appelé tête,* X *est un ensemble fini de variables ;* $\mathrm{lval} : V \to X$ *donne la variable affectée dans un nœud, et* $\mathrm{use} : V \to \mathcal{P}(X)$ *les variables dont l'affectation dépend*[9].

[9] $\mathcal{P}(X)$ dénote l'ensemble des parties de X.

Sorties

– un ensemble d'états de dépendances (définition 4)

Définition 4 (Etat de dépendances, adapté de la déf. 2 p. 408). *Un état de dépendances est composé de*

– *nœud : un nœud du CFG*
– *depgraph : un depgraph adapté (définition 5)*
– *pending : un pending adapté*

Habituellement, le graphe de dépendances collecte toutes les dépendances possibles entre affectations de variables sans distinguer les chemins d'exécution responsables de cette relation. Notre algorithme produit des états de dépendances potentiellement distincts par chemin d'exécution ; certains chemins peuvent donner le même état de dépendances.

Définition 5 (Graphe de dépendances adapté). *Comme les nœuds du CFG ne contiennent que des affectations, les nœuds du graphe de dépendances (auparavant des* DependencyNode*) sont réduits à un nœud du CFG. En effet pour un nœud n, le champ* ***id*** *vaut* lval(n) *et* ***index*** *vaut 0.*

Notre but dans cette section est de compter le nombre de solutions possibles, et donc pour éviter une approximation trop grossière, nous limitons les *pendings* aux données strictement nécessaires, c'est-à-dire l'ensemble des variables qui sont en attente de résolution.

Le *pending* de l'algorithme 1 est beaucoup plus riche car pour chaque variable en attente, il retient l'emplacement de la définition qui utilise cette variable. Ceci est utile pour des raisons de complexité en temps, qui ne nous concernent pas ici.

Algorithme adapté L'adaptation de l'algorithme 1 consiste à remplacer le dictionnaire *pendings* par un ensemble qui ne contient que ses clés.

Nous donnons dans l'algorithme 3 une version adaptée de l'algorithme 2 p. 420.

La proposition 1 assure que le nombre de solutions de l'algorithme initial est le même que pour l'algorithme adapté.

Proposition 1. *Etant donnés un CFG $\langle V, E, t, X, \text{lval}, \text{use}\rangle$, un nœud $n \in V$, et un état de dépendances adapté $E' = \langle n, D', P'\rangle$ obtenu par l'algorithme adapté, il existe un unique état de dépendances complet $E = \langle n, D, P\rangle$.*

Algorithme 3 : Analyse intra-bloc adaptée

Entrées : *cfg* $=\langle V, E, t, X, lval, use\rangle$, état
Sorties : Modification de état par effet de bord

```
1 si lval(état.nœud).est_une_clé(état.pendings) alors
2     nœud_définition = état.nœud ;
3     état.depgraph.ajouter_nœud(nœud_définition) ;
4     nœuds_utilisateur = {n ∈ V | lval(état.nœud) ∈ use(n) ∧
      ∀s ∈ succ_état.depgraph(n) , lval(s) ≠ lval(état.nœud)} ;
5     pour chaque nœud_utilisation ∈ nœuds_utilisateur faire
6         état.depgraph.ajouter_arc(nœud_utilisation,nœud_définition) ;
7     fin
8     état.pendings = état.pendings \ {lval(état.nœud)} ∪ use(état.nœud) ;
9 fin
```

Preuve

On veut imposer le même ordre de parcours du CFG aux deux algorithmes de telle sorte que cela établisse une séquence commune de traitement des nœuds du CFG. Nous allons vérifier que les modifications apportées par chaque traitement intra-bloc préservent la correspondance entre les états de dépendances original et adapté, ce qui garantira ce parcours commun et l'égalité du nombre de solutions.

Montrons que lors du traitement d'un nœud n, les *pendings* calculés dans l'algorithme original pour la variable en cours de traitement (c.-à-d. $lval(n)$) correspondent à l'ensemble calculé par l'algorithme adapté :

$$\{m \in V \mid lval(n) \in use(m) \wedge \forall s \in succ_{D'}(m)\,,\, lval(s) \neq lval(n)\}$$

Prouver cette égalité nécessite de mettre en regard les modifications apportées aux *pendings* avec celles apportées aux graphes de dépendances. La preuve peut se faire en vérifiant la double inclusion mais nous l'omettons ici car les notations la rendent fastidieuse. □

4.2 Majoration du nombre de solutions

Nous allons donner un majorant du cardinal de l'ensemble des solutions en fonction du CFG.

Proposition 2. *Le nombre de graphes de dépendances possibles, étant donné un CFG* $\langle V, E, t, X, lval, use\rangle$*, est majoré par*

$$2^{|V|^2+|X|} + |V|.2^{|V|^2}$$

Preuve

Dans l'algorithme 1, les états sont ajoutés à la solution à la ligne 16. Deux cas sont possibles : soit l'ensemble *pending* est vide, soit le nœud est la tête du CFG.

- Si *pending* est vide, chaque solution est déterminée par un nœud et un graphe de dépendances. Le nombre de solutions possibles est donc majoré par le produit du nombre de nœuds et du nombre de graphes de dépendances possibles, soit $|V|.2^{|V|^2}$ où V est l'ensemble des nœuds du CFG et donc contient ceux des graphes de dépendances.
- Si le nœud est la tête, chaque solution est déterminée par un graphe de dépendances et un ensemble *pending*. Le nombre de solutions est donc majoré par le produit du nombre de graphes de dépendances et du nombre de sous-ensembles de variables, soit $2^{|V|^2}.2^{|X|}$ où V est l'ensemble des nœuds du CFG et $|X|$ l'ensemble de variables.

  Le nombre de solutions est donc majoré par $2^{|V|^2+|X|} + |V|.2^{|V|^2}$.

□

4.3 Une famille de CFG ayant de nombreuses solutions

Soit *nsols* la fonction qui associe à un CFG $\langle V, E, t, X, \mathit{lval}, \mathit{use}\rangle$ le nombre de solutions (graphe de dépendance) distinctes retournées par l'algorithme 1.

En ordonnant les CFG par inclusion sur l'ensemble E, on remarque que la fonction *nsols* est croissante. En effet, ajouter un élément dans E ne peut pas enlever de chemin d'exécution. Par construction de l'ensemble des solutions, qui énumère des chemins d'exécution, le nombre de solutions ne peut que croître.

Le graphe complet est donc un pire cas. On se restreint donc maintenant à l'étude des graphes complets.

Il se trouve qu'en se limitant à une seule variable $x \in X$, c.-à-d. $|X| = 1$, et en considérant uniquement des affectations de la forme $x := x$ (pour tout $v \in V$, $\mathit{lval}(v) = x$ et $\mathit{use}(v) = \{x\}$), on obtient un nombre de solutions factoriel en le nombre de nœuds. Ceci peut se montrer par récurrence : il y a deux solutions pour le graphe complet à un nœud. Chaque ajout d'un nœud à un CFG multiplie le nombre de solutions par au moins $|V|$, par symétrie.

En utilisant la borne supérieure de la proposition 2, nous obtenons un encadrement du nombre de solutions retournées par l'algorithme : $2^{|V|log(|V|)} < \mathit{solutions} < 2^{|V|^2+|X|} + |V|.2^{|V|^2}$.

5 Résultats

Cette section décrit les résultats obtenus sur quelques cas concrets.

5.1 Malware : analyse d'arguments

Cet exemple est tiré d'un malware. Dans le but de ralentir l'analyse du code, les chaînes de caractères sont chiffrées[10]. Pour les utiliser le programme appelle une fonction de déchiffrement qui prend comme argument un pointeur sur la chaîne chiffrée (registre RDX), un pointeur sur une zone qui recevra la chaîne une fois déchiffrée (registre RCX) et la longueur de la chaîne (registre R8), voir la figure 16. Dans notre exemple, la fonction est référencée 49 fois. Pour découvrir toutes les chaînes de caractères en clair, il faudrait analyser toutes ces références et extraire les arguments [11].

```
                    loc_000000018000FE73
18000FE73  LEA       RDX, QWORD PTR [RIP+0xFFFFFFFFFFFF4366]
18000FE7A  LEA       RCX, QWORD PTR [RSP+0x50]
18000FE7F  MOV       R8D, 0x14
18000FE85  CALL      dec_str
```

Fig. 16.

Nous sommes ici dans le cas idéal pour utiliser l'algorithme. Voici quelques cas notables qui ont été résolus avec succès, et d'autres où l'algorithme donne un résultat qui sort de l'ordinaire. Tout d'abord, le traitement du cas donné en figure 16 renvoie :

```
0x18000fe85L : 0x14 0x1800041E0
'SHGetKnownFolderPath'
```

Ici, on voit que l'algorithme a trouvé la valeur finale de R8 (0x14) et du pointeur sur la chaîne de caractères chiffrée en mémoire RDX (0x1800041E0). Le lecteur averti aura noté que la chaîne de caractères déchiffrée apparaît dans la sortie : le script d'extraction émule également dans une *sandbox* l'appel à la fonction de déchiffrement et extrait cette dernière [12].

[10] Elles n'apparaissent donc pas en clair dans le binaire.

[11] Une référence peut utiliser plusieurs chaînes de caractères, par exemple dans un *if/then/else*.

[12] Tutorial Miasm SSTIC 2014 : `http://static.sstic.org/videos2014/SSTIC_2014-06-06_P09_Tutorial_Miasm.mp4`.

Autre exemple en figure 17. Ici, la difficulté réside dans le fait que la valeur de `R8D` n'est pas immédiate, elle est le résultat de l'addition de `RDI` et de `0x12` (dans le `LEA`). De plus, `R8D` est initialisé à zéro grâce au `XOR EDI, EDI` présent un peu plus haut. Grâce au moteur de simplification de Miasm, ce `XOR` est transformé en `EDI = 0`.

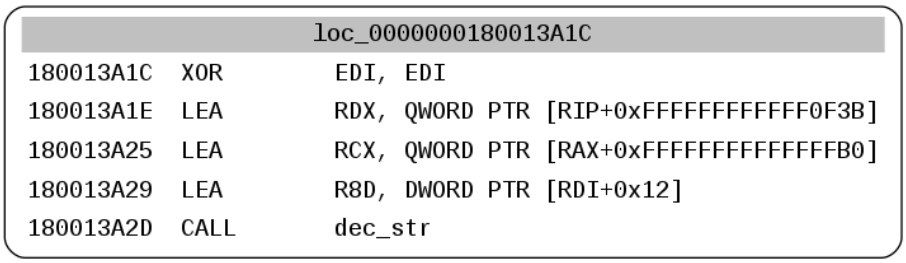

Fig. 17.

Le résultat donne ici :

```
0x180013a2dL : 0x12 0x180004960
'www.earthtools.org'
```

Dans l'exemple de la figure 18, l'initialisation se fait à travers un `OR` logique avec 0xFFFFFFFF. Ici, on utilise de surcroît le fait que ce `OR` est transformé en une simple affectation.

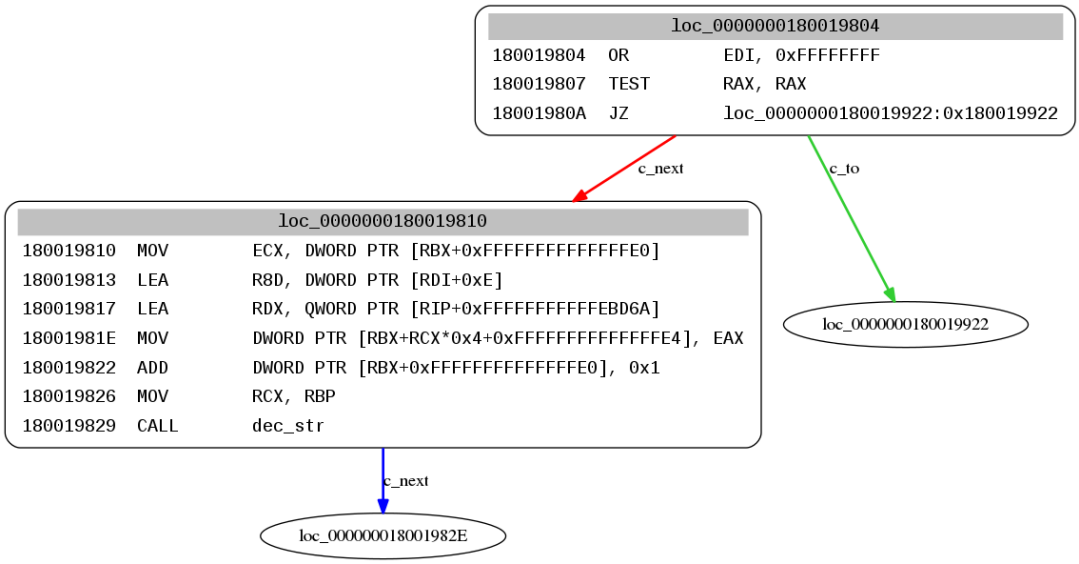

Fig. 18.

Le résultat :

```
0x180019829L : 0xD 0x180005588
'cryptbase.dll'
```

Voyons maintenant les cas de la figure 19 où le suivi n'a pas donné une constante.

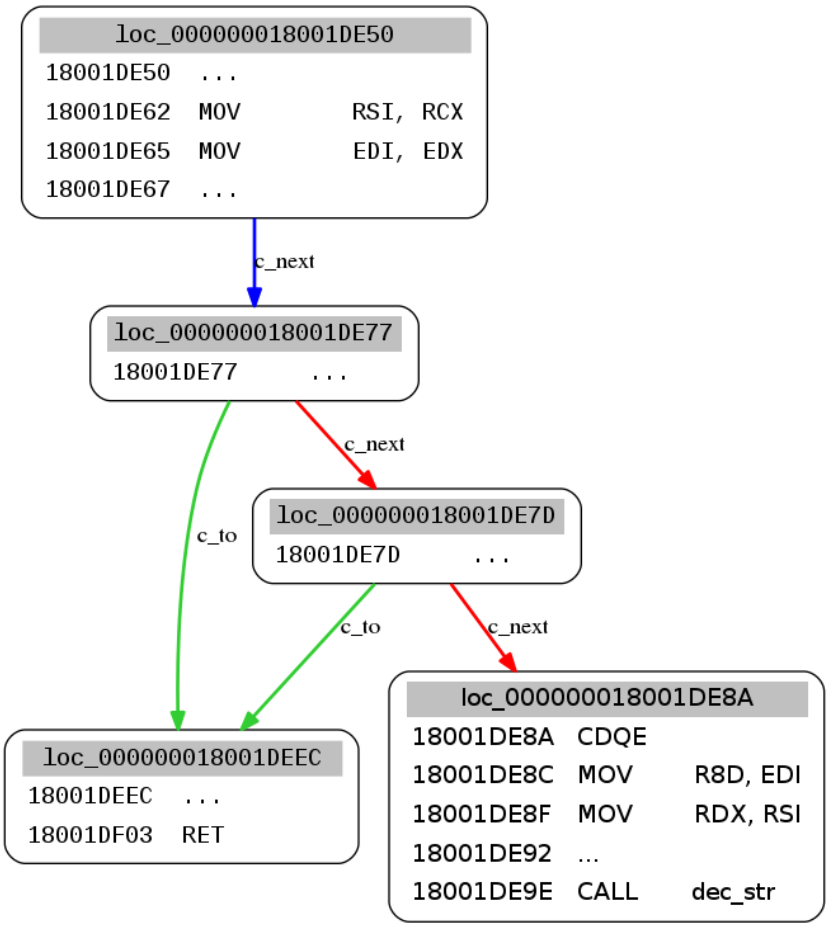

Fig. 19.

Ici l'algorithme remonte les variables jusqu'au point d'entrée de la fonction. On se rend compte que les variables dépendent finalement directement des arguments de cette dernière.

```
0x18001de9eL : {RDX_init[0:32],0,32, 0x0,32,64} RCX_init
ERROR: Not Imm
```

L'expression décrivant la taille se lit : «`RDX` initial dans lequel seul les 32 premiers bits de poids faible ont été conservés».

Si l'on veut poursuivre l'analyse, il faudrait relancer l'algorithme avec les nouvelles dépendances obtenues sur la fonction parente.

5.2 Switch/cases

L'exemple qui suit n'est pas tiré d'un cas réel mais permet d'illustrer un cas intéressant [13]. Pour plus de clarté, le code source C du programme est repris ci-dessous. Nous allons bien entendu lancer l'algorithme sur la version binaire[14]. Le but est ici d'identifier toutes les valeurs possibles de la variable *ret* à la fin de la fonction *test*. En assembleur, nous allons demander un suivi de dépendances du registre `EAX` au moment du `RET`.

```
int test(unsigned int argc, char** argv)
{
        unsigned int ret;
        if (argc == 0)
                ret = 0x1001;
        else if (argc < 2)
                ret = 0x1002;
        else if (argc <= 5)
                ret = 0x1003;
        else if (argc != 7 && argc*2 == 14)
                ret = 0x1004;
        else if (argc*2 == 14)
                ret = 0x1005;
        else if (argc & 0x30)
                ret = 0x1006;
        else if (argc + 3 == 0x45)
                ret = 0x1007;
        else
                ret = 0x1008;
        return ret;
}

int main(int argc, char** argv)
{
        return test(argc, argv);
}
```

Listing 3. Switch/case

Cette analyse peut être réalisée en lançant le plugin IDA fourni dans Miasm[15]. Il nous indique que la valeur de retour peut prendre les valeurs allant de 0x1001 à 0x1008. En y ajoutant les dépendances implicites (voir

[13] Il est tiré des codes d'exemples de Miasm : `https://github.com/cea-sec/miasm/blob/master/example/samples/simple_test.c`.

[14] Version binaire : `https://github.com/cea-sec/miasm/blob/master/example/samples/simple_test.bin`.

[15] Plugin IDA *DepGraph* : `https://github.com/cea-sec/miasm/blob/master/example/ida/depgraph.py`.

la section 5.3) et un solveur de contraintes, on obtient des valeurs de *argc* permettant d'atteindre chacune de ces sorties :

- `EAX = 0x1001` avec `argc = 0x0` ;
- `EAX = 0x1002` avec `argc = 0x1` ;
- `EAX = 0x1003` avec `argc = 0x5` ;
- `EAX = 0x1004` avec `argc = 0x80000007` ;
- `EAX = 0x1005` avec `argc = 0x7` ;
- `EAX = 0x1006` avec `argc = 0x11` ;
- `EAX = 0x1007` avec `argc = 0x42` ;
- `EAX = 0x1008` avec `argc = 0x40000005`.

Il existe également des cas où l'algorithme renvoie un surensemble des solutions possibles. Prenons le code suivant (pour des raisons de lisibilité, le code proposé est en langage C) :

```
b = 0;
if (a)
   b |= 0x11220000;
if (a)
   b |= 0x3344;
```

Listing 4. Sur-ensemble

Ici, la condition des *if* est la même, et a n'est pas modifié entre les deux tests. Après exécution de ce code, nous savons que b vaut soit `0x0`, soit `0x11223344`. Néanmoins, l'algorithme d'analyse des dépendances ne prend pas en compte le fait que les tests sont liés et renverra le produit cartésien des chemins empruntés. Ceci donnera l'ensemble `0x0`, `0x11220000`, `0x3344`, `0x11223344`. Ici, nous avons bien nos deux solutions légitimes, plus deux solutions aberrantes.

Il existe au moins deux façons de résoudre ce problème. Nous pouvons détecter ce genre de motifs dans le graphe de flot d'exécution et *factoriser* la condition, ce qui donnera le code :

```
b = 0;
if (a) {
   b |= 0x11220000;
   b |= 0x3344;
}
```

Listing 5. Code factorisé

Ici, on retombera sur deux solutions uniques.

Une autre possibilité est d'utiliser un solveur de contraintes sur les conditions à remplir pour honorer le chemin d'exécution emprunté dans

une solution donnée. Le but ici est de voir si ces dernières ne génèrent pas des cas non satisfaisables. Sur nos quatre solutions précédemment trouvées, deux sont impossibles à réaliser, et on retombe sur nos deux solutions, `0x0` et `0x11223344`.

5.3 Suivi de données stockées dans la pile

La plupart des architectures et des programmes utilisent intensivement la pile, et stockent donc des variables en mémoire. Nous l'avons vu, le suivi de dépendance des cases mémoire dans le cas général est difficile. Néanmoins, dans un programme classique on peut considérer que les accès à la pile sont réalisés à travers l'utilisation du registre de pile, ou à travers l'utilisation d'un registre découlant du registre de pile et défini dans ladite fonction. Typiquement dans un programme compilé pour l'architecture *x86*, les données dans la pile seront accédées en utilisant ESP ou EBP. De plus en observant le prologue de la fonction courante, on peut voir que le registre *EBP* découle de *ESP*.

```
int main(int argc, char** argv)
{
        volatile int buffer[0x100];
        buffer[10] = 0x1337beef;
        return buffer[10];
}
```

Listing 6. Fonction à analyser

```
1 push        ebp
2 mov         ebp, esp
3 sub         esp, 0x400
4 mov         dword ptr [ebp+0xfffffc28], 0x1337beef
5 mov         eax, dword ptr [ebp+0xfffffc28]
6 leave
7 ret
```

Listing 7. Fonction à analyser

Dans la suite, on note *ESP_init* la valeur symbolique du registre de pile à l'entrée de la fonction. Pour réaliser notre analyse finale, nous allons procéder à une réécriture du programme pour remplacer tous les accès à la pile par des accès exprimés par une expression basée sur *ESP_init*, la valeur du registre ESP au début de la fonction.

Pour cela nous avons besoin de deux choses :

– déterminer à chaque ligne du code assembleur de la fonction la hauteur de la pile par rapport à *ESP_init* ;

– déterminer de la même façon la valeur de EBP[16] par rapport à *ESP_init*.

Ceci nous ramène à un nouveau problème : comment déterminer la hauteur de la pile en tout point dans une fonction donnée ? Ce problème nécessitant à lui seul une nouvelle présentation SSTIC, nous nous baserons pour le moment sur un oracle : IDA propose la fonction *GetSpd(adresse)* qui renvoie, pour une adresse de ligne assembleur donnée, la différence de hauteur de pile par rapport au point d'entrée de la fonction comportant cette ligne. Sur la figure 20, on peut voir la traduction de la fonction 7 en langage intermédiaire. Sur la figure 21, la même portion de code mais dont les registres ESP et EBP ont été réécrits en se basant sur *ESP_init*.

```
main
0  ESP = ESP_init

1  @32[(ESP+0xFFFFFFFC)] = EBP

2  ESP = (ESP_init+0xFFFFFFFC)

3  EBP = ESP

4  @32[(EBP+0xFFFFFC28)] = 0x1337BEEF

5  EAX = @32[(EBP+0xFFFFFC28)]

6  EBP = @32[EBP]

7  ESP = ESP_init

8  ESP = (ESP+0x4)
8  IRDst = @32[ESP]
```

Fig. 20. IR original

```
main
0  ESP = ESP_init

1  @32[(ESP_init+0xFFFFFFFC)] = EBP_init

2  ESP = (ESP_init+0xFFFFFFFC)

3  EBP = (ESP_init+0xFFFFFFFC)

4  @32[(ESP_init+0xFFFFFC24)] = 0x1337BEEF

5  EAX = @32[(ESP_init+0xFFFFFC24)]

6  EBP = @32[(ESP_init+0xFFFFFFFC)]

7  ESP = ESP_init

8  ESP = (ESP_init+0x4)
8  IRDst = @32[ESP_init]
```

Fig. 21. IR réécrit

Nous pouvons lancer l'algorithme d'analyse de dépendance de EAX à la fin de la fonction, qui donnera le résultat figure 22. Il faut noter que l'algorithme ne fonctionne pas pour le moment (c'est une de ses limites) dans le cas où l'on écrit *partiellement* les cases mémoire. Pour résoudre ce problème, on peut réécrire le programme en remplaçant les accès mémoire ayant lieu sur une plage mémoire par plusieurs accès mémoire unitaires (sur un octet). On retombe alors dans le cas précédent où l'algorithme suivra uniquement les dépendances nécessaires.

[16] Ou tout autre registre alias de la pile.

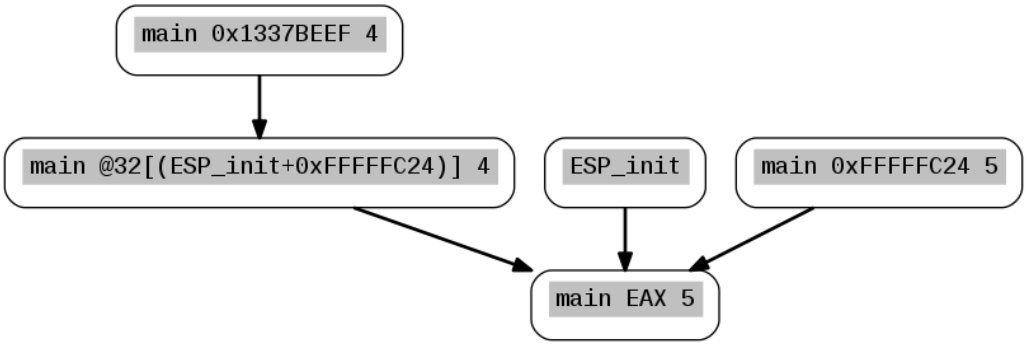

Fig. 22.

Explicite/Implicite Pour définir la dépendance de donnée, nous avons précédemment utilisé le terme dépendance explicite. Si nous considérons le graphe de la figure 23, nous voyons que a vaut $b+c$, qui vaudra finalement soit `0x3`, soit `0x4`.

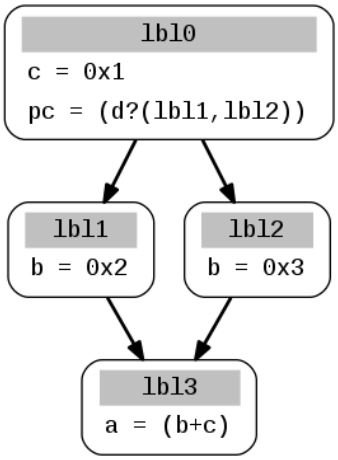

Fig. 23.

Cette valeur dépend de la branche empruntée par le code. On peut donc dire que cette valeur dépend également de la condition de cette branche, c'est-à-dire de la valeur de d. Cette dépendance est notée dépendance *implicite*. L'algorithme pourra être modifié pour prendre en compte ces dépendances : une fois l'analyse intra-bloc effectuée, on peut lier toutes les variables en cours de résolution à la valeur de PC, ce qui créera naturellement un lien *implicite* vers toutes les variables impliquées dans le saut conditionnel. Cela soulève d'autres problèmes, notamment l'explosion

du nombre de variables à suivre, ainsi qu'une difficulté certaine à interpréter le graphe résultat.

Fig. 24. Le petit poucet prend possession de l'outil d'analyse de dépendance, permettant une plus grande vitesse d'analyse

5.4 Performances

En section 4.3 nous avons vu un cas complexe où l'algorithme retourne un nombre élevé de solutions : dans le cas d'un graphe complet de quelques *basic blocks*, il ne finira pas dans un temps raisonnable.

Pour analyser les performances dans un cas concret, cette section décrit les résultats obtenus sur l'analyse d'un malware public dont le SHA256 est `f7aee92474bbd1d3e118943c804ece476f7a2485d469edfb5e4cd06717f8a2eb` [17].

On peut voir que les chaînes de caractères utilisées par le binaire (nom de DLL, clef de base de registre, ...) sont chiffrées. De même que dans les exemples décrits en section 5.1, ces chaînes sont déchiffrées à la volée par une fonction. Cette fonction prend en argument trois paramètres sur la pile. Le premier argument correspond à un identifiant de la chaîne de caractères à déchiffrer. Il faudra donc dé-aliaser la pile, en utilisant *IDA* comme heuristique (voir 5.3).

[17] Référence sur Malwr.com : `https://malwr.com/analysis/YTczNDdmODk1ZWRmNGEyMWIxMWI0MTkwY2QyYTAxNDM/`

La fonction est à l'adresse `0x100079B0` et a 26 références. Le script d'analyse va suivre le premier argument dans chacune de ces références. L'analyse va ici couvrir la plupart des cas introduits dans cet article.

Le script est lancé sur une machine portable standard, sur un seul processeur de type *Intel i7*. Quelques valeurs représentatives :

- temps total : 7.73s ;
- nombre de solutions concrètes trouvées : 28 ;
- nombre de solutions indéterminées (dans une boucle) : 1 ;
- répartition désassemblage et traduction IR/de-aliasing/depgraph/émulation : 47.91%/45.94%/3.85%/1.39%

Un test de performance a également été réalisé sur une fonction de taille raisonnable[18], en analysant le registre de pile `esp` en fin de fonction, dans le but de stresser l'analyseur. Les temps obtenus sont du même ordre de grandeur.

6 Conclusion

L'algorithme présenté dans cet article, le *DependencyGraph*, se place entre deux familles d'analyses existantes :

- l'analyse statique du flot de données sans tenir compte des chemins, qui donne un résultat même sur des graphes complexes, mais qui renvoie une solution imprécise ;
- l'analyse du flot de données *PathSensitive*, qui donne des solutions précises, mais qui peut tomber dans l'explosion combinatoire.

Il a pour but de raffiner les résultats obtenus grâce à la première famille, tout en limitant le coût algorithmique de la seconde. Ainsi, il est suffisamment précis et rapide pour servir de brique intermédiaire dans plusieurs algorithmes, notamment en recherche de vulnérabilités ou en analyse de malware.

Même s'il souffre de quelques problèmes classiques de dépendance de donnée, comme les alias de variables, il est possible d'utiliser des méthodes pour pouvoir travailler sur de vrais exemples.

De plus, il peut être couplé à des heuristiques pour répondre à des questions plus complexes comme la découverte automatique de prototypes de fonctions ou le suivi d'arguments inter-fonctions. Des travaux sont notamment en cours pour définir et rendre utilisable une version comprenant les dépendances implicites, et pour peut-être introduire l'algorithme inverse, l'*InfluenceGraph*, s'il s'avère intéressant.

[18] Fonction à l'adresse 0x10001030

Références

1. Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. pages 1083–1094, 2014.
2. Fabrice Desclaux. Miasm : Framework de reverse engineering. *https://www.sstic.org/media/SSTIC2012/SSTIC-actes/miasm_framework_de_reverse_engineering/SSTIC2012-Article-miasm_framework_de_reverse_engineering-desclaux_1.pdf*, 2012.
3. Thomas Dullien. Reil : A platform-independent intermediate representation of disassembled code for static code analysis. *https://static.googleusercontent.com/media/www.zynamics.com/en//downloads/csw09.pdf*, 2009.
4. Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3) :319–349, 1987.
5. Christian Hammer. Information flow control for java - a comprehensive approach based on path conditions in dependence graphs. July 2009. ISBN 978-3-86644-398-3.
6. Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1) :26–60, January 1990.
7. Ranjit Jhala and Rupak Majumdar. Path slicing. *SIGPLAN Not.*, 40(6) :38–47, June 2005.
8. Ken Kennedy. Use-definition chains with applications. *Computer Languages*, 3(3) :163–179, 1978.
9. Ken Kennedy. *A survey of data flow analysis techniques*. IBM Thomas J. Watson Research Division, 1979.
10. Florent Marceau. Utilisation du data tainting pour l'analyse de logiciels malveillants. *https://www.sstic.org/media/SSTIC2009/SSTIC-actes/Utilisation_du_data_tainting_pour_l_analyse_de_log/SSTIC2009-Article-Utilisation_du_data_tainting_pour_l_analyse_de_logiciels_malveillants-marceau.pdf*, 2009.
11. Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *SIGPLAN Not.*, 19(5) :177–184, 1984.
12. Frank Tip. A survey of program slicing techniques. *Journal of programming languages*, 3(3) :121–189, 1995.
13. Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.
14. Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7) :446–452, 1982.

A Code *BackwardSlice* d'Angr

```
from argparse import ArgumentParser

import angr # version 4.6.2.25
from miasm2.core.graph import DiGraph


# Parse arguments
parser = ArgumentParser()
parser.add_argument("filename", help="Target binary")
parser.add_argument("addr", help="Target basic block address")
parser.add_argument("-s", "--statement", type=int, default=-1,
                    help="Statement number in IR")
parser.add_argument("-o", "--output", default="graph.dot",
                    help="Output graph in dot")
args = parser.parse_args()

# Run Angr
project = angr.Project(args.filename, load_options={'auto_load_libs':False})
cfg = project.analyses.CFG(context_sensitivity_level=2, keep_state=True)
cdg = project.analyses.CDG(cfg)
ddg = project.analyses.DDG(cfg)
addr = int(args.addr, 0)

target_node = cfg.get_any_node(addr)
bs = project.analyses.BackwardSlice(cfg, cdg=cdg, ddg=ddg,
                                    targets=[(target_node, args.statement)])

# Highlight result in dot
class HighlightedCFG(DiGraph):

    def __init__(self, project, *args, **kwargs):
        super(HighlightedCFG, self).__init__(*args, **kwargs)
        self.project = project

    def node2lines(self, node):
        """
        Returns an iterator on cells of the dot @node.
        A DotCellDescription or a list of DotCellDescription are accepted
        @node: a node of the graph
        """
        vex = self.project.factory.block(node.addr).vex
        to_highlight = self.highlight[node.addr]
        for i, line in enumerate(vex.statements):
            attr = {}
            if i in to_highlight:
                attr["bgcolor"] = "red"
            yield [self.DotCellDescription(text=str(i), attr={}),
                   self.DotCellDescription(text=str(line), attr=attr)]
        yield[self.DotCellDescription(text=str(i + 1), attr={}),
              self.DotCellDescription(text="NEXT: %s" % vex.next, attr={})]

graph = HighlightedCFG(project)
graph.highlight = bs.chosen_statements

for node in cfg.graph.nodes():
    graph.add_node(node)
for src, dst in cfg.graph.edges():
    graph.add_edge(src, dst)

open(args.output, "w").write(graph.dot())
```

Pour lancer ce script sur l'exemple de 2.3 :

```
$ python backwardslice.py --statement 2 test.bin 0x401016 && xdot
    graph.dot
```

B Code *PathGroup* d'Angr

```
from argparse import ArgumentParser

import angr # version 4.6.2.25

# Parse arguments
parser = ArgumentParser()
parser.add_argument("filename", help="Target binary")
parser.add_argument("addr", help="Target basic block address")
args = parser.parse_args()

# Run Pathgroup
addr = int(args.addr, 0)
project = angr.Project(args.filename, load_options={'auto_load_libs'
    :False})

state = project.factory.entry_state()
state = project.factory.blank_state(addr=0x401000)

# Activating veritesting result in a crash
path_group = project.factory.path_group(state, veritesting=False)
path_group = path_group.explore(find=(addr,))

# Display results
founds = path_group.found
for i, found in enumerate(founds):
    print "Path %d" % i
    print "\tEAX: %s" % found.state.regs.eax
    print "\tEBX: %s" % found.state.regs.ebx
```

Pour lancer ce script sur l'exemple de 2.3 :

```
$ python pathgroup.py test.bin 0x401016
Path 0
    EAX: <BV32 Reverse((1 + reg_8_0_8) .. Reverse(reg_8_1_32)[23:0])
        >
    EBX: <BV32 0x1>
```

C Code *BackTrace* de Metasm

```
# Version 2c9871c

require 'metasm'
include Metasm

def sliceit(filename, offset, reg)

    puts 'Open binary...'
    pe = PE.decode_file(filename)
    dasm = pe.disassemble_fast_deep 'entrypoint'
    puts "Dasm ok, slice on %#x" % offset
    reg = reg.to_sym

    # Run backtrace
    log = []
    dasm.debug_backtrace = 1
    ret = dasm.backtrace(reg, offset, :log => log)
    puts "Result"
    p ret
    puts "OK"

    # return the trace
    log
end


filename = ARGV[0]
slice_addr = ARGV[1].to_i(16)
reg = ARGV[2]

slice = sliceit(filename, slice_addr, reg)
# slice is an array which contains backtrace logs
slice.each { |ev, *args|
    if ev == :di    # decodedinstruction
        after = args[0]
        before = args[1]
        instr = args[2]
        puts "#{instr} [#{before} -> #{after}]"
    end
}
```

Pour lancer ce script sur l'exemple de 2.3 :

```
$  ruby dasm_pe.rb test.bin 0x401018 eax
Open binary...
Dasm ok, slice on 0x401018
backtracking  eax from 401018h ret for
  backtrace 401016h mov eax, ebx  eax => ebx
  backtrace up 401016h->401014h  ebx
  backtrace 40100eh mov ebx, ecx  ebx => ecx
  backtrace up 40100ch->40100ah  ecx
  bt loop at 401014h: ebx => ecx (401014h)
  backtrace 401000h mov ecx, 1  ecx => 1
backtrace  found 1 from 401000h mov ecx, 1 orig
  backtrace result: 1
Result
[Expression[1]]
OK
401016h mov eax, ebx [eax -> ebx]
40100eh mov ebx, ecx [ebx -> ecx]
401000h mov ecx, 1 [ecx -> 1]
```