HAL
open science

Taint-Based Return Oriented Programming

Colas Le Guernic, François Khourbiga

To cite this version:

Colas Le Guernic, François Khourbiga. Taint-Based Return Oriented Programming. SSTIC 2018 - Symposium sur la sécurité des technologies de l'information et des communications, Jun 2018, Rennes, France. pp.1-30. hal-01848575

HAL Id: hal-01848575

https://inria.hal.science/hal-01848575v1

Submitted on 24 Jul 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Taint-Based Return Oriented Programming

Colas Le Guernic[1,2] et François Khourbiga[3]
colas.le-guernic@intradef.gouv.fr
francois.khourbiga@orange.com

[1] DGA Maîtrise de l'Information
[2] Univ. Rennes, Inria, CNRS, IRISA
[3] Orange Cyberdéfense

Résumé. Dans ce papier nous présentons une nouvelle approche pour la recherche de gadgets. Plutôt que d'utiliser une sémantique symbolique précise et coûteuse, nous nous basons sur une sémantique de teinte qui prend en compte correctement les registres, la pile, et la mémoire. Cette approche a été implémentée dans un outil libre, T-Brop, qui démontre son intérêt et son positionnement intermédiaire : plus rapide que les outils symboliques et permettant des requêtes plus expressives que les outils syntaxiques.

1 Introduction

Lors d'une évaluation de sécurité sur un logiciel, l'analyste doit émettre un avis et des recommandations en terme d'architecture, de configuration, et de correction d'éventuelles failles identifiées. À défaut de preuve formelle de sécurité, l'analyste doit être compétent et utiliser des outils à l'état de l'art afin d'émettre un avis pertinent. Par ailleurs, ses recommandations auront plus de poids s'il réussit à démontrer la criticité d'une faille par un code d'exploitation capable de contourner les contre-mesures prévues.

C'est dans ce contexte que nous avons développé l'approche *taint-based return oriented programming* et l'outil associé, T-Brop, qui facilite la réalisation de ROP chains. Outre la correction de la faille associée, cet outil nous permet d'insister sur la nécessité de mettre en place des contre-mesures modernes, comme la vérification d'intégrité du flot d'exécution (*control flow integrity*, CFI [1]), afin d'entraver ce type d'attaques.

La faille classique du débordement de tampon sur la pile (*stack buffer overflow* [15]) permet d'injecter du code dans la mémoire d'un processus et de détourner le flot d'exécution vers ce code injecté via l'écrasement de l'adresse de retour. Si ce type de failles persiste encore, leur exploitation directe est heureusement aujourd'hui rendue difficile par différentes contre-mesures. Parmi celles-ci, la prévention de l'exécution des données (*data*

execution prevention, DEP, ou *write xor execute*, W⊕X) empêche, comme son nom l'indique, l'exécution d'une zone mémoire modifiable : une zone mémoire est modifiable ou exécutable mais pas les deux.

Tant qu'une contre-mesure de ce type est active, il n'est pas possible d'injecter et exécuter un code malveillant. Il reste cependant possible de détourner le flot d'exécution vers du code déjà présent en mémoire, on parle alors d'attaque en réutilisation de code (*code reuse attack* [17]).

Parmi ces approches, le ROP (*return oriented programming* [14, 24, 29]) consiste en l'enchainement de courtes séquences d'instructions terminant par un retour de fonction ou plus généralement tout branchement indirect. Ces portions de code sont appelées *gadget*, et leur enchainement une *ROP chain*. Des contre-mesures, notamment les différentes variantes de CFI, permettent de détecter ou entraver, dans une certaine mesure, les ROP chains. Encore faut-il qu'elles soient correctement mises en œuvre.

Dans le cadre d'un débordement de tampon sur la pile, plutôt que d'injecter du code, les adresses des gadgets de la ROP chain sont placées sur la pile. Lors de l'exécution du retour de fonction en fin de gadget, l'adresse du gadget suivant est récupérée sur la pile, provoquant son exécution et permettant ainsi l'enchainement des gadgets de la ROP chain.

De nombreux outils ont été développés pour faciliter l'élaboration de ROP chains. Comme nous le verrons plus loin, ces outils peuvent être classés en deux catégories distinctes suivant la précision de l'analyse effectuée sur chaque gadget. Les plus rapides se limitent à la syntaxe : la liste des instructions. Les plus puissants calculent une relation symbolique entre entrées et sorties. Ces derniers sont cependant très (trop) lents.

Nous proposons une approche intermédiaire basée sur une sémantique imprécise : les flux explicites d'information. Cette approche permet un filtrage plus puissant que les méthodes purement syntaxique, tout en étant plus rapide que les approches symboliques.

Notre contribution principale est un algorithme efficace pour calculer les relations de dépendances des entrées vers les sorties de chaque gadget, en particulier les dépendances vers le pointeur d'instruction à la fin du gadget, que nous appelons condition de chaînage. Ces relations de dépendances nous permettent d'évaluer la complexité des gadgets plus pertinemment qu'avec leur seule taille, et peuvent être exploitées pour filtrer les gadgets plus efficacement qu'avec des expressions régulières. Nous décrivons également comment obtenir une approximation de l'ensemble des dépendances atteignables par l'ensemble des ROP chains dérivables à partir d'un ensemble donné de gadgets. Nous avons mis en œuvre notre approche dans l'outil libre T-Brop : `https://github.com/DGA-MI-SSI/T-Brop`.

Dans le reste du papier nous commençons par présenter succinctement les principaux outils disponibles dédiés à l'élaboration de ROP chain en section 2. Nous faisons ensuite un rappel sur les matrices booléennes en section 3 Nous nous en servons pour présenter le cœur théorique de notre contribution en commençant par une analyse offrant des garanties fortes de correction sections 4, et une autre moins conservative section 5. Nous montrons ensuite comment utiliser ce calcul de dépendances pour constituer et interroger une base de gadgets section 6. Notre implémentation est brièvement décrite section 7 et nos premiers résultats expérimentaux sont rapportés section 8. Pour finir nous évoquons quelques pistes d'améliorations et travaux futurs section 9 avant de conclure section 10.

2 État de l'art

L'élaboration d'une ROP chain nécessite plusieurs étapes. La première consiste à lister les gadgets disponibles Pour ce faire, on commence par identifier les branchements indirects. On remonte ensuite de quelques instructions à partir de chacun de ces branchements pour former des gadgets. On recherche ensuite des gadgets en fonction de l'effet souhaité, pour initialiser un contexte par exemple. La dernière étape consiste en l'assemblage des gadgets sous une forme garantissant leur enchainement. Cette dernière étape peut nécessiter une recherche de gadgets supplémentaires.

De nombreux outils sont dédiés à la réalisation de ROP chain. La table 1 en liste quelques-uns. Pour chacun d'eux nous avons récupéré quelques statistiques en terme de popularité et d'activité le 7 avril 2018. La popularité n'est qu'une mesure imprécise de la qualité ou l'intérêt d'un outil, et les dates de premier et dernier commit sur la branche *master*, ou même leur nombre, ne reflète pas nécessairement le degré d'activité d'un projet. Par exemple, rp++ n'a connu qu'une douzaine de commits depuis fin 2012, et près de la moitié concerne la documentation. Dans ce cas précis, cette faible activité s'explique par la robustesse et le périmètre limité de l'outil. Par ailleurs, certains projets listés sont en fait des framework intégrant une fonctionnalité d'aide à la réalisation de ROP chain : radare2, Pwntools, PEDA, BARF, ida-sploiter. Les différentes statistiques ne concernent pas uniquement la fonctionnalité ROP et sont donc sur-évaluées. Angr[4] [30] et Metasploit[5] proposent également une aide à l'élaboration de ROP chain, mais dans un projet distinct : angrop et rex-rop_builder (msfrop) respectivement.

[4] angr, a binary analysis framework, `http://angr.io/`
[5] RAPID7 metasploit, `https://www.metasploit.com/`

name	watch	star	fork	commit first	commit last	commit count	URL
radare2 /R	413	7119	1421	2009	-	17775	`github.com/radare/radare2`
Pwntools	247	3740	767	2013	-	3290	`github.com/Gallopsled/pwntools`
PEDA	163	2446	453	2012	-	94	`github.com/longld/peda`
ROPgadget	103	1377	333	2011	2017	422	`github.com/JonathanSalwan/ROPgadget`
BARF	69	977	134	2014	-	816	`github.com/programa-stic/barf-project`
rp++	56	573	123	2012	2017	138	`github.com/0vercl0k/rp`
mona	68	540	211	2013	2017	133	`github.com/corelan/mona`
Ropper	43	535	107	2014	-	582	`github.com/sashs/Ropper`
rop-tool	47	474	98	2014	-	179	`github.com/t00sh/rop-tool`
ropc	36	222	24	2012	2013	16	`github.com/pakt/ropc`
angrop	18	188	37	2014	2017	141	`github.com/salls/angrop`
universalrop	18	178	23	2017	-	10	`github.com/kokjo/universalrop`
roputils	17	156	46	2014	2016	195	`github.com/inaz2/roputils`
rop_compiler	14	145	29	2015	2017	212	`github.com/jeffball55/rop_compiler`
ropa	3	142	26	2017	-	271	`github.com/orppra/ropa`
xrop	10	136	35	2014	2017	82	`github.com/acama/xrop`
ROPER	12	132	19	2016	-	326	`github.com/oblivia-simplex/roper`
ida-sploiter	14	111	27	2014	2017	10	`github.com/iphelix/ida-sploiter`
ropeme	7	95	38	2013	2016	4	`github.com/packz/ropeme`
ropstone	7	79	9	2016	2017	10	`github.com/blasty/ropstone`
nrop	13	63	15	2014	2016	126	`github.com/awailly/nrop`
Agafi	16	61	19	2014	2015	13	`github.com/CoreSecurity/Agafi`
ropc-llvm	10	61	14	2013	2013	2	`github.com/programa-stic/ropc-llvm`
DrGadget	10	46	10	2014	2017	19	`github.com/patois/DrGadget`
EasyROP	6	44	11	2016	2017	55	`github.com/uZetta27/EasyROP`
ropchain	10	43	12	2014	2015	93	`github.com/SQLab/ropchain`
rop-chainer	3	40	14	2016	-	11	`github.com/wizh/rop-chainer`
Mov2Rop	1	3	1	2017	2017	66	`github.com/0xEval/bsc-thesis`
rex-rop_builder	37	1	0	2011	2017	29	`github.com/rapid7/rex-rop_builder`
PSHAPE	N.A.	N.A.	N.A.	2016	2016	1	`sites.google.com/site/exploitdevpshape`

Tableau 1. Quelques outils d'aide à la création de ROP-chain. Données en date du 7 avril 2018. Dans la colonne *last commit*, '-' signifie qu'il y a eu au moins un commit sur la branche *master* depuis le 1 janvier 2018.

2.1 Filtrage syntaxique

Tous ces outils permettent de lister l'ensemble des gadgets jusqu'à une certaine taille dans un exécutable. Certains se limitent au gadgets terminant par un retour de fonction, ou ne gèrent pas tout le jeu d'instruction. La limite en terme de taille s'exprime parfois en bytes ou en nombre d'instructions. Cela peut expliquer une certaine variabilité dans le nombre de gadgets retournés.

À partir de cette liste, l'analyste peut rechercher le gadget souhaité. S'il n'est pas directement dans la liste des gadgets disponibles, l'analyste aura recours à des expressions régulières de plus en plus permissives. Par exemple, si elle ne trouve pas directement le gadget `mov rax, rbx ; ret`, elle commencera par rechercher les gadgets du type :

```
mov rax, rbx
.*
ret
```

Si cette nouvelle requête ne permet pas de trouver le gadget souhaité, il sera nécessaire d'étendre la recherche pour obtenir des gadgets différents dont la sémantique reste similaire au gadget initial. Par exemple :

```
xchg rax, rbx
.*
ret
```

```
push rbx
.*
pop rax
.*
ret
```

Afin de trouver des gadgets satisfaisant l'effet recherché, il peut falloir construire un nombre considérable d'expressions régulières, ou templates syntaxiques. Afin d'éviter de trop nombreux faux positifs, il faut potentiellement écrire des templates complexes. Par exemple, le dernier template pourrait être raffiné pour avoir autant de `push` que de `pop` entre le `push rbx` et le `pop rax`, et aucune modification de `rax` entre le `pop rax` et le `ret`.

2.2 Filtrage symbolique

Pour un filtrage plus efficace, certains outils calculent une relation symbolique précise entre les sorties et les entrées de chaque gadget. La recherche de gadgets équivalants peut se faire grâce à des règles de réécriture [5], ou de façon plus systématique avec un solveur SMT (*Satisfiability Modulo Theories* [2]) comme dans le module DEPlib 2.0 [31] du débogueur Immunity[6], OptiRop [22], nrop, BARFgadgets [12], ou PSHAPE [7].

[6] `http://www.immunityinc.com/products/debugger/`

Ainsi, l'analyste peut trouver tous les gadgets équivalant à un gadget donnée en effectuant une requête SMT pour chaque gadget identifié. Cette requête est construite à partir des expressions symboliques représentant la sémantique des deux gadgets comparés. Pour gagner en flexibilité, l'équivalence n'est recherchée que pour les écritures du gadget cible.

Résoudre toutes ses requêtes SMT pour chaque recherche de gadget est très coûteux, et parfois trop précis. En effet, quand l'analyste recherche un `mov rax, rbx`, en général c'est pour initialiser le registre `rax` à partir du registre contrôlé `rbx`. Dans ce cas, toute affectation de l'image de `rbx` par une fonction inversible au registre `rax` conviendrait. Par exemple, l'instruction `lea rax, [rbx+1]` n'est pas sémantiquement équivalente à `mov rax, rbx`. Pourtant cette instruction permet de choisir arbitrairement la valeur de `rax` si l'analyste contrôle la valeur de `rbx`.

Plutôt que de chercher une équivalence, il est donc parfois plus utile de chercher des dépendances. On peut les trouver facilement en cherchant les lectures qui apparaissent dans l'exression symbolique de l'écriture souhaitée, après un éventuel pré-traitement pour éviter les faux positifs. Ici, on cherche tous les gadgets tels que l'expression de `rax` dépende de `rbx`. C'est l'approche prise par angrop :

```
    lea rax, [rbx+1]
    ret
```

```
Stack change: 0x8
Changed registers: set(['rax'])
Popped registers: set([])
Register dependencies:
rax: [rbx ()]
```

2.3 Classification de gadgets

On l'a vu, la recherche de gadget passe par la rédaction de templates syntaxiques ou sémantiques qui peut être fastidieuse et demander une certaine expertise. Certains outils fournissent des templates pour classer les gadgets dans des catégories simples et utiles : affectations, lecture ou écriture en mémoire, sur la pile...

Ces gadgets élémentaires peuvent être combinés, parfois automatiquement, pour former des ROP chains élémentaires qui seront utilisées comme des gadgets. Par exemple, si on cherche à déréférencer un registre $r2$ pour mettre le résultat dans $r1 = [r2]$ et qu'aucun gadget élémentaire ne permet de réaliser cette action, on peut chercher un gadget réalisant la même opération sur d'autres registres $r3 = [r4]$, et des gadgets permettant d'obtenir les affectations $r4 = r2$ et $r1 = r3$. En combinant ces trois gadgets, élémentaires ou non, on obtient le déréférencement désiré.

Ces combinaisons se font sous l'hypothèse que l'analyste contrôle les données pointées par le registre de pile. Il arrive que les données contrôlées soient pointées par un autre registre. Il faut donc d'abord modifier le registre de pile pour qu'il pointe vers les données contrôlées, c'est ce qu'on appelle un *stack pivot*.

2.4 Génération de ROP chain

Si suffisamment de gadgets, ou combinaisons de gadgets, ont été trouvés dans chaque catégorie, certains outils, comme mona ou ROPgadget, proposent de générer automatiquement une ROP chain particulière : un appel à `VirtualProtect` ou `execve` par exemple.

D'autres outils, comme Gadget Exploit Framework [24], Q [28], ropc, ou ropc-llvm [11], sont plus ambitieux et proposent de compiler un programme arbitraire vers une ROP chain. Cependant ces outils génèrent en général des ROP chains très longues et ne sont que des preuves de concept utilisables uniquement sur des exemples bien choisis.

Pour tous ces outils, la génération d'une ROP chain est conditionnée par le présence d'un certain nombre de gadgets prédéfinis. Une approche exhaustive [25] permettrai la création de ROP chains dans plus de contextes mais l'explosion combinatoire associée semble insurmontable.

2.5 Approches exotiques

Quelques outils prennent une approche différente. Plutôt que de calculer une relation symbolique pour chaque gadget, Agafi [6] exécute les gadgets dans une machine virtuelle avec un contexte de départ bien choisi. Le filtrage sur les gadgets se fait en vérifiant des requêtes logiques sur les états d'entrée et de sortie obtenus. Il est ainsi possible de filtrer tous les gadgets tels que, sur la trace exécutée, `rax` en sortie est égal à `rbx` en entrée. Agafi permet également de générer des ROP chains avec un contexte cible.

Mov2Rop [3] quant à lui se base sur M/o/Vfuscator [4] pour transformer un shellcode arbitraire en une séquence d'instructions `mov`. Il ne reste plus qu'à chercher des gadgets pour chacun des `mov` obtenus.

Pour finir ROPER [9] (à ne pas confondre avec Ropper) utilise la programmation génétique pour générer des ROP chains.

2.6 Conclusion

La plupart des outils automatiques échouent s'ils ne trouvent pas de gadgets dans les catégories dont ils ont besoin. L'analyste doit alors

chercher lui même des gadgets et un enchainement à partir d'un listing de séquences d'instructions avec ou sans information symbolique.

Les outils purement syntaxique sont très rapide mais nécessite un travail de filtrage par expression régulière fastidieux. Les outils symboliques permet un filtrage plus puissant mais nécessitent un calcul symbolique et pour certains la résolution d'un grand nombre de requêtes SMT. En partique, ils ne sont pas satisfaisants, dixit Tavis Ormandy [20, 21] :

> « Did OptiROP ever get released ? I always solve these constrained ROP problems manually because no good tools exist. »

> « I know about DEPLIB, but it's not really usable. I think I'm just going to have to write the tool I want :(»

3 Matrices de Boole

Avant de décrire notre approche, quelques rappels sur l'algèbre de Boole à deux éléments et la manipulation de matrices sont nécessaires.

L'algèbre de Boole à deux éléments peut être définie comme l'ensemble des deux valeurs de vérité *Vrai* et *Faux*, que l'on notera 1 et 0 respectivement, muni des opérateurs suivants :

- la disjonction, c'est à dire le *ou* logique, parfois notée $\vee$ et que nous désignerons également par $+$;
- la conjonction, c'est à dire le *et* logique, parfois notée $\wedge$ et que nous désignerons également par $\times$;
- et la négation, dont nous n'aurons pas l'usage.

Pour les puristes : les notations $+$ et $\times$ sont abusives car $(\{0,1\}, +, \times)$ n'est qu'un demi-anneau, mais elles nous permettrons de définir le produit matriciel avec des notations plus habituelles.

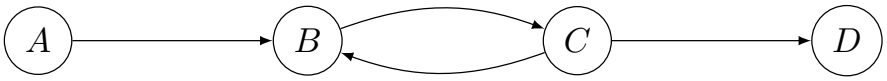
Une matrice est un tableau à double entrée. Pour une matrice M, l'entrée à la ligne i, colonne j, est notée $m_{i,j}$. Pour deux matrices A et B, si n, le nombre de colonnes de A, est égal au nombre de lignes de B, on définit leur produit $M = AB$ de la façon suivante :

$$m_{i,j} = \sum_{k=0}^{n-1} a_{i,k} b_{k,j}$$

Nous nous intéressons en particulier aux matrices carrées (autant de lignes que de colonnes) à valeur dans $\{0,1\}$. Ces matrices sont très utiles pour représenter des relations entre éléments d'un même ensemble.

Une relation est un ensemble de couples, deux éléments x et y sont dit en relation si et seulement si le couple (x, y) appartient à cet ensemble. Nous nous en servirons pour représenter une relation de dépendance entre registres, plus précisément la présence, ou non, d'un flux d'information entre une entrée et une sortie lors de l'exécution d'un gadget. Usuellement, ces matrices booléennes servent à représenter la présence, ou non, d'une arrête entre deux nœuds d'un graphe : la matrice d'adjacence.[7] Il y a équivalence entre matrices booléennes, graphes orientés, et relations.

Exemple 1. Sur un ensemble à quatre éléments A, B, C, et D, la relation définie par les couples (A, B), (B, C), (C, B), et (C, D) peut être représentée sous forme de graphe, de matrice, ou de tableau.



$$M = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

out \ in	A	B	C	D
A	·	·	·	·
B	↲	·	↲	·
C	·	↲	·	·
D	·	·	↲	·

Les 0 sont volontairement grisés dans la représentation matricielle pour améliorer sa lisibilité. Les étiquettes de la représentation tabulaire seront tronquées si nécessaire par la suite.

Le produit matriciel se définit également pour les matrices booléennes. Ici, l'opérateur d'addition est une disjonction, et l'opérateur produit, une conjonction. Ainsi les éléments d'une matrice $M = AB$ sont définit comme des disjonctions de conjonctions :

$$m_{i,j} = \bigvee_{k=0}^{n-1} a_{i,k} \wedge b_{k,j}$$

Autrement dit $m_{i,j}$ vaut 1 si il existe k tel que $a_{i,k}$ et $b_{k,j}$ valent tous les deux 1. Dans le cas contraire $m_{i,j}$ vaut 0. Quant à la somme, elle est définie par $M = A + B$ comme une disjonction composante par composante.

En considérant la matrice d'adjacence M d'un graphe : M représente tous les chemins de longueur 1 et M^k représente les chemins de longueur exactement k. Par extension M^0 est la matrice identité notée I avec des 1 uniquement sur la diagonale, et représente les chemin de longueur 0. $I + M$ représente les chemin de longueur au plus 1.

[7] Nous utilisons la convention $m_{i,j} = 1$ si et seulement si il existe une transition du nœud j au nœud i, contrairement à ce qui se fait d'habitude.

4 Sémantique de teinte

Plutôt que de se baser uniquement sur la syntaxe ou sur une sémantique précise, symbolique, nous prenons une approche intermédiaire basée sur une sémantique de teinte [26, 27]. L'objectif est d'être plus rapide que les approches symboliques et plus informatif qu'une approche purement syntaxique pour un analyste qui cherche a étendre son influence sur l'état du système. Concrètement, on teinte les registres influencés, et on propage ensuite cette teinte avec les règles habituelles.

Plus formellement (mais pas trop), l'état e du système est une fonction qui à chaque registre ou adresse mémoire associe sa valeur. Suite au déclenchement d'une vulnérabilité, un analyste exerce une influence sur un registre r si et seulement si il est capable de choisir la valeur du registre r, parmi au moins deux options. Cette influence sur l'état du système est représentée par un vecteur v de valeurs de vérité indiquant pour chaque registre s'il est influencé ou non. La même notion d'influence peut être définie pour la mémoire, mais oublions la pour le moment.

4.1 Matrice de dépendances

Cette notion d'influence initiale, au déclenchement de la vulnérabilité, est secondaire, l'analyste peut choisir lui même quels registres il estime pouvoir influencer. Ce qui nous intéresse ici est l'évolution de cette influence.

Étant donné une séquence d'instruction s, on appelle matrice de dépendance de s, notée M_s, la matrice dans l'algèbre de Boole représentant l'évolution de l'influence par s. De façon similaire à l'étude des flux d'information (*information flow* [10]) ou l'analyse d'atteignabilité des variables (*reachability analysis* [18]), la valeur de $M_s(i, j)$ est égale à 1 si et seulement si il existe deux états identiques partout sauf sur le registre j tels que l'application de la séquence d'instruction s mène à deux états pour lesquels le registre i est différent. De façon équivalente, si quelque soit l'état initial, changer la valeur du registre j en entrée ne change pas la valeur du registre i en sortie, alors j n'influence pas i, et réciproquement. À partir d'un vecteur d'influence v, et après la séquence d'instructions s, on obtient le vecteur d'influence $M_s v$.

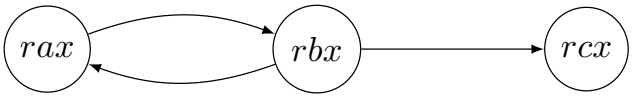
Exemple 2. En se limitant aux registres rax, rbx, et rcx, une influence initiale sur le registre rbx uniquement est représentée par le vecteur $v = \begin{pmatrix} 0 & 1 & 0 \end{pmatrix}^\top$. Considérons le gadget suivant et sa matrice de dépendances :

```
    xchg rax, rbx
    xor rcx, rcx
    add rcx, rax
    jmp rcx
```

$$M = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

out \ in	rax	rbx	rcx
rax	·	↲	·
rbx	↲	·	·
rcx	·	↲	·

ou sous forme de graphe :



L'influence après l'exécution de cette séquence est donnée par le produit $Mv = (1\ 0\ 1)^\top$, soit une influence sur rax et rcx mais pas rbx.

Le calcul de la matrice M est indécidable dans le cas général. Ici, nous ne nous intéressons qu'à des séquences d'instructions sans structure de contrôle, le problème devient alors NP-complet (voir annexe A). On ne peut donc pas calculer M de façon exacte en un temps raisonnable, mais on peut facilement en calculer une approximation conservative en considérant chaque instruction individuellement, comme les algorithmes classiques de propagation de teinte. Cette approximation est dite conservative, car si elle peut rajouter des influences (faux positifs), elle ne peut pas en oublier (faux négatifs). Si M_g est la matrice d'un gadget g et M_{op} la matrice d'une instruction précédent g, alors $M_g M_{op}$ est la matrice du gadget $op; g$.

Exemple 3. En reprenant la séquence d'instructions de l'exemple 2 on a :

```
g0:
    jmp rcx
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}_{\text{g0}}$$

```
g1:
    add rcx, rax
    jmp rcx
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}_{\text{g0}} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}_{\texttt{add}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}_{\text{g1}}$$

```
g2:
    xor rcx, rcx
    add rcx, rax
    jmp rcx
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}_{\text{g1}} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}_{\texttt{xor}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}_{\text{g2}}$$

```
g3:
    xchg rax, rbx
    xor rcx, rcx
    add rcx, rax
    jmp rcx
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}_{\text{g2}} \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}_{\texttt{xchg}} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}_{\text{g3}}$$

Ainsi le calcul de M peut être mutualisé pour les gadgets ayant un suffixe commun. Le coût de l'extension d'un gadget est limité au produit de deux matrices booléennes creuses. En pratique la complexité est linéaire en le nombre de registres suivis. On remarquera que la matrice de dépendance d'une instruction ne modifiant pas les registres est la matrice identité.

4.2 Condition de chaînage

Ces matrices de dépendances peuvent être utilisées pour filtrer un ensemble de gadgets en sélectionnant uniquement ceux autorisant un flux entre deux registres précis par exemple. Une autre approche consiste à définir un vecteur u représentant un registre cible (ou une disjonction de registres cibles) ; à partir du vecteur d'influence v et du gadget g on peut influencer le registre cible seulement si $u^\top M_g v$ est 1.

Exemple 4. À partir de rbx et en appliquant le gadget de l'exemple 2 on ne peut pas influencer la valeur d'rbx en sortie $\begin{pmatrix}0 & 1 & 0\end{pmatrix}$, mais on peut (peut-être, car on calcul une approximation conservative) influencer celle d'rcx, $\begin{pmatrix}0 & 0 & 1\end{pmatrix}$. En effet :

$$\begin{pmatrix}0 & 1 & 0\end{pmatrix}\begin{pmatrix}0 & 1 & 0\\ 1 & 0 & 0\\ 0 & 1 & 0\end{pmatrix}\begin{pmatrix}0\\ 1\\ 0\end{pmatrix} = 0 \qquad \begin{pmatrix}0 & 0 & 1\end{pmatrix}\begin{pmatrix}0 & 1 & 0\\ 1 & 0 & 0\\ 0 & 1 & 0\end{pmatrix}\begin{pmatrix}0\\ 1\\ 0\end{pmatrix} = 1$$

Un registre cible particulièrement intéressant est le pointeur d'instruction. Si u est le vecteur dont la seule valeur à 1 correspond au pointeur d'instruction, alors, pour un gadget g, on définit la condition de chaînage de g, notée u_g^{cc}, le vecteur $u^\top M_g$. Par définition, u_g^{cc} représente l'ensemble des registres qui peuvent influencer la valeur du pointeur d'instruction après l'exécution du gadget g. Pour pouvoir chaîner g avec d'autres gadgets il faut contrôler au moins l'un de ces registres, et idéalement tous.

Exemple 5. La condition de chaînage du gadget de l'exemple 2 est le vecteur $\begin{pmatrix}0 & 1 & 0\end{pmatrix}$, ou sous forme tabulaire :

out \ in	rax	rbx	rcx
rip	·	↲	·

Ainsi, on ne pourra pas contrôler le chaînage de ce gadget si on ne contrôle pas rbx avant son exécution.

Cette caractérisation des gadgets par une matrice de dépendance et une condition de chaînage peut être étendue aux séquences de gadgets. Ainsi, si le gadget $g0$ est caractérisé par M_0 et u_0^{cc}, et le gadget g_1 par M_1 et u_1^{cc}, alors la matrice de dépendance de la séquence $g_0; g_1$ est $M_1 M_0$. La ligne correspondant au pointeur d'instruction est $u_1^{cc} M_0$, mais il faut également pouvoir contrôler u_0^{cc} pour permettre l'enchaînement de ces deux gadgets. Ainsi nous définissons la condition de chaînage de $g_0; g_1$ par $u_1^{cc} M_0 + u_0^{cc}$, où $+$ est ici un *ou* logique composante par composante.

4.3 Superposition de gadgets

Outre la composition de gadgets, la matrice de dépendances se prête bien à la superposition de gadgets, c'est à dire à l'exécution non-déterministe d'un gadget parmi un ensemble donné.

La matrice de dépendances d'un ensemble de gadgets G est une approximation conservative de la matrice de dépendances de chacun des gadgets de G. Elle se calcule facilement avec la formule suivante :

$$M_G = \sum_{g \in G} M_g$$

M_G représente donc l'ensemble des flux que l'on peut espérer en exécutant un gadget de G. M_G^n représente les flux que l'on peut espérer en exécutant exactement n gadget de G à la suite. Et $(I + M_G)^n$ représente ceux que l'on peut espérer en exécutant au plus n gadget de G.

Par ailleurs, $(I + M_G)^n$ converge rapidement, en au plus la dimension de M_G et donc du nombre de registres suivis. Ainsi, on peut obtenir efficacement avec une exponentiation rapide une sur-approximation de l'ensemble des flux que l'on peut espérer avec toute ROP chain de longueur quelconque construite avec des gadgets de G. Si le flux qui nous intéresse n'apparaît pas dans $(I + M_G)^\infty$, il est absolument inutile d'essayer de construire une ROP chain. Il faut alors chercher d'autres gadgets.

4.4 Traitement de la mémoire

Contrairement aux registres les flux d'information impliquant la mémoire dépendent fortement du contexte d'exécution. En effet une lecture sur rax n'est pas une lecture sur rbx, quel que soit le contexte, mais un déréférencement de rax peut-être équivalent à un déréférencement de rbx si ces deux registres pointent sur une même case mémoire. Or dans le cadre de la recherche de gadgets nous ne connaissons pas le contexte d'exécution.

Afin de garantir le calcul d'une approximation conservative nous devons donc prendre en compte toutes les situations d'aliasing potentielles.

Nous traitons donc la mémoire grâce à un pseudo-registre *mem* auquel correspondent une ligne et une colonne dans notre matrice de dépendance. Toute lecture de la mémoire propage ses dépendances. Toute écriture dans la mémoire introduit de nouvelles dépendances, mais sans écraser les anciennes : *mem* en sortie dépend toujours au moins de *mem* en entrée.

Exemple 6. Considérons maintenant les registres *rax*, *rbx*, *rcx*, et *mem*.

```
mov [rax], rbx
pop rcx
ret
```

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix}$$

out \ in	rax	rbx	rcx	mem
rax	↲	·	·	·
rbx	·	↲	·	·
rcx	↲	↲	·	↲
mem	↲	↲	·	↲

Le registre *rcx* en sortie dépend de *rax*, *rbx*, *mem*, et aussi de *rsp* non représenté ici pour des raisons de place. En effet, à partir d'un état du système quelconque, il est possible de modifier *rcx* en sortie en modifiant uniquement la valeur pointée par *rsp*, ou en modifiant *rax* ou *rsp* pour qu'ils soient égaux, ou encore, s'ils sont égaux, en modifiant *rbx*.

Afin d'adhérer à notre définition d'influence, nous propageons également les dépendances liées aux registres déréférencés. Cela semble particulièrement légitime pour les lecture en l'absence de contexte précis. Après l'instruction `mov rbx, [rax]`, *rbx* dépend de la mémoire mais aussi de *rax*. Il peut être également utile de savoir quels registres sont déréférencés. C'est pourquoi nous introduisons également un pseudo-registre *deref* qui dépend des registres déréférencés et de lui même. Aucun registre ne dépend jamais de *deref*. Le calcul des dépendances vers *deref* n'est pas purement syntaxique et suit les règles de propagation que nous venons de décrire.

Exemple 7. Considérons maintenant les registres *rax*, *rbx*, et *deref*.

```
mov rax, [rbx]
xchg rax, rbx
mov [rax], rbx
ret
```

$$M = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

out \ in	rax	rbx	drf
rax	·	↲	·
rbx	·	↲	·
drf	·	↲	↲

Le registre *deref* en sortie ne dépend que de *rbx* (et *deref*). En effet, lors du second déréférencement *rax* a la valeur de *rbx* en entrée.

5 Compromis correction/utilité

Nous venons d'introduire une approche permettant de calculer efficacement une approximation des matrices de dépendances et conditions de chaînage d'un ensemble de gadgets. Elle offre des garanties sur l'absence de faux négatifs : aucune dépendance ne manque. Cette exigence forte de correction nous oblige à prendre en compte toutes les situations d'aliasing potentiel et introduit également des faux positifs, en particulier à cause de la propagation pour toute lecture sur la mémoire des dépendances de toutes les écritures qui la précèdent.

Afin d'obtenir une vision plus réaliste des dépendances exploitables, il nous semble légitime de relâcher cette exigence de correction [16].

5.1 mem_r et pep_r

La première étape consiste à casser ce lien entre écritures et lectures sur la mémoire. Au lieu d'un seul pseudo-registre mem, nous considérons deux registres mem_r et mem_w représentant respectivement les lectures et écritures sur la mémoire. Toute lecture sur la mémoire dépend de mem_r et des éventuels registres déréférencés, et mem_r ne dépend que de mem_r. Toute écriture sur la mémoire introduit une dépendance vers mem_w, et seul mem_w dépend de mem_w.

L'analyste a accès à toutes les dépendances sauf celles passant directement par la mémoire. Ces dernières sont toujours accessibles si besoin : un registre qui dépend de mem_r est potentiellement influencé par toutes les dépendances de mem_w. Cette flexibilité à un coût car l'ordre entre écritures et lectures est perdue. Une lecture ne dépend plus uniquement des écritures qui la précèdent mais de toutes les écritures du bloc.

Exemple 8. En reprenant l'exemple 6 avec les (pseudo-)registres rax, rbx, rcx, mem_r, et mem_w.

```
mov [rax], rbx
pop rcx
ret
```

out \ in	rax	rbx	rcx	m_r	m_w
rax	↲	·	·	·	·
rbx	·	↲	·	·	·
rcx	·	·	·	↲	·
m_r	·	·	·	↲	·
m_w	↲	↲	·	·	↲

Le registre rcx en sortie ne dépend plus que de mem_r (et rsp non représenté ici). L'analyste peut retrouver les dépendances potentielles en observant la ligne correspondant à mem_w : le contenu de la mémoire, dont dépend rcx, est potentiellement influencé par rax et rbx.

5.2 Représentation de la pile

Séparer lectures et écritures retire également les dépendances passant par une partie fort utile de la mémoire : la pile. Nous allons donc la traiter différemment du reste de la mémoire.

Comme précédemment notre approche consiste à ajouter des registres :

- $stack_0$ représente le haut de la pile ;
- $stack_1$, $stack_2$, …, $stack_n$, représentent les n valeurs suivantes ;
- $stack_{under}$ représente les tréfonds de la pile, l'agrégat de toutes les cellules sous $stack_n$, et garde toujours une dépendance sur lui même. Plus exactement on sépare ce registre en deux, comme pour la mémoire, en $stack_{under_r}$ et $stack_{under_w}$;
- on représente également les valeurs au dessus de la pile avec $stack_{-1}$, $stack_{-2}$, …, $stack_{-m}$, $stack_{over_r}$ et $stack_{over_w}$.

La position dans la pile représentée par chacun de ces registres est relative au pointeur de pile. En particulier le pointeur de pile pointe toujours vers $stack_0$. Lors d'un `push` ou d'un `pop` par exemple, le pointeur de pile ne pointe pas vers un autre registre, c'est plutôt la valeur de dépendance de tous les pseudo-registres de pile qui est décalée. Ces registres nous permettent de traiter précisément les opérations simples sur la pile, de type `push`, `pop`, `call` et `ret`. Nous traitons également les additions et soustractions d'une (petite) constante au pointeur de pile : la matrice de dépendance est générée à la volée en fonction de la constante ajoutée, ou soustraite, en gérant correctement les désalignements de pile.

Exemple 9. Pour simplifier on ne sépare pas ici la mémoire en deux. On considère les registres rax, $stack_{over}$, $stack_{-1}$, $stack_0$, $stack_1$, $stack_2$, et $stack_{under}$. Considérons le gadget :

```
push rax
ret
```

Chacune de ces instructions présente les dépendances suivantes :

`ret`	rax	s_{over}	s_{-1}	s_0	s_1	s_2	s_{und}
rax	↲	·	·	·	·	·	·
s_{over}	·	↲	↲	·	·	·	·
s_{-1}	·	·	·	↲	·	·	·
s_0	·	·	·	·	↲	·	·
s_1	·	·	·	·	·	↲	·
s_2	·	·	·	·	·	·	↲
s_{und}	·	·	·	·	·	·	↲

`push rax`	rax	s_{over}	s_{-1}	s_0	s_1	s_2	s_{und}
rax	↲	·	·	·	·	·	·
s_{over}	·	↲	·	·	·	·	·
s_{-1}	·	↲	·	·	·	·	·
s_0	↲	·	·	·	·	·	·
s_1	·	·	·	↲	·	·	·
s_2	·	·	·	·	↲	·	·
s_{und}	·	·	·	·	·	↲	↲

Après un `ret` le haut de la pile, $stack_0$, contient la valeur qui était immédiatement sous l'adresse de retour, $stack_1$, ce qui explique la dépendance de $stack_1$ vers $stack_0$. De même, après un `push rax`, la valeur initialement en haut de la pile, $stack_0$, se retrouve juste sous la valeur qui vient d'être poussée, ce qui explique la dépendance de $stack_0$ vers $stack_1$.

La matrice de dépendance du gadget `push rax`, `ret` est, comme précédemment, le produit des matrices des instructions qui le compose :

$$\begin{pmatrix} 1&0&0&0&0&0&0\\ 0&1&1&0&0&0&0\\ 0&0&0&1&0&0&0\\ 0&0&0&0&1&0&0\\ 0&0&0&0&0&1&0\\ 0&0&0&0&0&0&1\\ 0&0&0&0&0&0&1 \end{pmatrix}_{\texttt{ret}} \begin{pmatrix} 1&0&0&0&0&0&0\\ 0&1&0&0&0&0&0\\ 0&1&0&0&0&0&0\\ 1&0&0&0&0&0&0\\ 0&0&0&1&0&0&0\\ 0&0&0&0&1&0&0\\ 0&0&0&0&0&1&1 \end{pmatrix}_{\texttt{push rax}} = \begin{pmatrix} 1&0&0&0&0&0&0\\ 0&1&0&0&0&0&0\\ 1&0&0&0&0&0&0\\ 0&0&0&1&0&0&0\\ 0&0&0&0&1&0&0\\ 0&0&0&0&0&1&1\\ 0&0&0&0&0&1&1 \end{pmatrix}$$

Ce qui donne sous forme tabulaire :

out \ in	rax	s_{over}	s_{-1}	s_0	s_1	s_2	s_{und}
rax	↲	·	·	·	·	·	·
s_{over}	·	↲	·	·	·	·	·
s_{-1}	↲	·	·	·	·	·	·
s_0	·	·	·	↲	·	·	·
s_1	·	·	·	·	↲	·	·
s_2	·	·	·	·	·	↲	↲
s_{und}	·	·	·	·	·	↲	↲

La cellule immédiatement au-dessus de la pile dépend[8] de rax. Par ailleurs, $stack_2$, la troisième cellule de la pile, dépend du fond de la pile (qui dépend également de $stack_2$). Cette interdépendance est due au fait que $stack_{under}$ est un accumulateur. Au moment du `push`, $stack_2$ est poussé dans $stack_{under}$. Lors du `ret`, la valeur en haut de $stack_{under}$ est remontée vers $stack_2$. Mais $stack_{under}$ est un accumulateur et on ne sait pas ce qu'il y en haut, d'où l'interdépendance. Quant à la condition de chaînage, elle est initialisée à $stack_0$ par le `ret` et transformée en rax par le `push`.

En dehors des ces opérations simples, toute modification du pointeur de pile provoque une corruption de la pile : mem_w prend toutes les dépendances de la pile et toutes les cellules de la pile dépendent de mem_r et des dépendances du pointeur de pile.

Avec $stack_{over}$, $stack_0$, $stack_{under}$, mem_r et mem_w, cela donne :

[8] La conservation des valeurs stockées au-dessus de la pile n'est pas garantie, surtout en mode noyau. Dans ce cas on n'utilisera pas les registres *stack* avec index négatif.

out \ in	s_{over}	s_0	s_{und}	m_r	m_w
s_{over}	·	·	·	↲	·
s_0	·	·	·	↲	·
s_{und}	·	·	·	↲	·
m_r	·	·	·	↲	·
m_w	↲	↲	↲	·	↲

$$\begin{pmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \end{pmatrix}$$

Exemple 10. Considérons le gadget :

```
    mov rsp, rdx
    push rax
    pop rbx
    pop rcx
    ret
```

Notre approche calcule en particulier que rbx ne dépend que de rax, et que rcx dépend de mem_r et rdx. Le calcul matriciel est laissé en exercice.

Contrairement à l'introduction de mem_r et mem_w, à partir de laquelle il est possible de retrouver une approximation conservative, comme expliqué en section 5.1, certaines dépendances potentielles lié à des alias sur la pile sont définitivement perdues. En effet, pour un registre qui ne dépend pas directement de l'état de la pile, on ne peut pas savoir si ses dépendances sont passées par la pile ou non, et si elles ont pu être modifiées en transit.

Exemple 11. Considérons le gadget suivant :

```
    push rax
    mov [rbx], rcx
    pop rax
    ret
```

Avec notre approximation de la pile, rax en sortie ne dépend que de rax en entrée. Pourtant si au moment du déréférencement rbx pointe vers le haut de la pile alors rax prend la valeur de rcx. D'après notre définition de l'influence, rax est donc influencé par rax, rbx, et rcx.

Angrop aussi ignore l'alias éventuel :

```
Changed registers: set(['rax'])
Popped registers: set([])
Register move: [rax to rax, 64 bits]
Register dependencies:
    rax: [rax ()]
Memory write:
    address (64 bits) depends on: ['rbx']
    data (64 bits) depends on: ['rcx']
```

6 Base de gadgets

6.1 Construction

T-Brop construit une base de gadgets contenant les informations de dépendances et les conditions de chaînage. La première étape est classique et consiste en la recherche de toutes les adresses correspondant à un retour de fonction ou un saut dynamique. Ces adresses servent de point d'entrée pour la construction des gadgets.

Comme illustré par l'exemple 3, la construction des gadgets, avec leur matrice de dépendances et condition de chaînage, se fait de manière itérative. Pour tout gadget en cours de construction on en insère d'abord une copie dans notre base de gadget et on l'étend en rajoutant une instruction au début. Pour les architectures ayant des instructions de taille variable, plusieurs extensions sont possibles. Autant de copies que nécessaire sont alors créées.

La plupart des approches considèrent les gadgets jusqu'à une certaine taille, en byte ou en nombre d'instructions. Au moment de l'extension d'un gadget nous avons sa matrice de dépendance et condition de chaînage. Nous pouvons utiliser ces informations pour calculer une fonction de coût et décider si le gadget doit être étendu ou s'il est déjà assez complexe. Notre fonction de coût est une combinaison linéaire du :

- taux d'augmentation du nombre de dépendances : $\max(0, (n - d/d))$, où n est le nombre de 1 dans la matrice de dépendances, et d le nombre de (pseudo-)registres. Ainsi pour une permutation ce critère est nul.
- nombre de dépendances dans la condition de chaînage.
- nombre de registres déréférencés. Trois déréférencements dont l'adresse dépend d'un seul et même registre comptent pour un, un déréférencement dont l'adresse dépend de trois registres compte pour trois.
- nombre d'instructions.

Exemple 12. Considérons les deux gadgets suivants :

```
add rbx, [rdx+4*rcx]
imul rbx
call [rax+0xC05FEFE]
```

```
push rax
push rbx
push rcx
mov rcx, rbx
mov rbx, rax
pop rax
pop rdx
pop rsi
pop rdi
ret 0x10
```

La gadget de gauche est plus court en nombre d'instructions ainsi qu'en nombre de bytes (13 contre 16). Pourtant dans le cadre d'une recherche de gadgets il nous semble plus raisonnable d'étendre le second.

6.2 Consultation

À partir de l'ensemble des gadgets étudiés, une base de données associant entre autre à tout couple de registres (i, j) l'ensemble des gadgets g tels que $M_g(i, j) = 1$, est créée.

Une fois la base constituée il est possible d'effectuer un filtrage pour en extraire une liste de gadgets intéressants, triés par coût croissant. Cette liste réduite peut alors être étudiée par une approche plus classique, comme pour les outils purement syntaxiques, ou passée à un outil automatique plus précis. Le filtrage peut se faire sur la présence, l'unicité, ou l'absence d'une ou plusieurs dépendances, sur la condition de chaînage...

Le filtrage sur les dépendances nous permet également de filtrer les gadgets par rapport au décalage sur la pile. Cela peut être utile à un analyste qui souhaiterait éviter une région de la pile qu'il ne contrôle pas. Si $stack_i$ ne dépend que de $stack_j$ alors à priori il y a un décalage de $i - j$ cellules ; si $stack_i$ ne dépend que de $stack_j$ et $stack_{j+1}$ alors on peut borner le décalage entre $i - j$ et $i - j - 1$ cellules ; Le décalage peut également être majoré ou minoré en considérant les dépendances vers $stack_{under}$ et $stack_{over}$. Les bornes calculées avec cette heuristique sont correctes en général, mais il est toujours possible d'effectuer une permutation sur la pile sans changer le pointeur de pile.

Le filtrage sur les dépendances a été étendu aux combinaisons de deux gadgets réalisant un flux d'un registre $r1$ à un registre $r2$. Pour le premier gadget nous commençons par éliminer tous ceux dont la condition de chaînage n'est pas réalisable. Ensuite pour chaque registre $r3$, le gadget le moins coûteux réalisant une dépendance de $r1$ vers $r3$ est recherché. Pour le second gadget, pour chaque $r3$, nous éliminons tous ceux dont la condition de chaînage après l'exécution du premier gadget n'est pas réalisable. Parmi ces gadgets nous retournons le moins coûteux parmi ceux réalisant $r3$ vers $r2$.

Exemple 13. Dans le cadre d'une analyse, une vulnérabilité nous permet de contrôler un buffer pointé par rax et la cible du prochain saut. On cherche d'abord un gadget ayant une dépendance de rax et mem_r vers rsp : aucun résultat. On cherche donc à construire une chaîne passant par un registre intermédiaire, on obtient :

```
mov rcx, rax
call [rax+8]

mov rdx, [rcx+0x50]
mov rbp, [rcx+0x18]
mov rsp, [rcx+0x10]
jmp rdx
```

Bien sûr l'outil ne regarde que les dépendances, d'autres propositions contenant par exemple `mov rdx, [rax-0x30]` et `movsxd rsp, [rdx-0x50]` sont inutiles dans notre contexte : nous ne contrôlons pas `[rax-0x30]` et nous voulons un contrôle total sur *rsp*.

L'outil n'est pas magique, mais permet à l'analyste d'éliminer efficacement un grand nombre de gadgets inutiles et de mettre en avant certains gadgets qui n'auraient pas forcément été considérés avec un simple grep.

7 Implémentation

L'approche a été implémentée en python 3 dans l'outil T-Brop disponible sur le compte GitHub des divisions SSI de la DGA[9]. Pour le désassemblage nous nous appuyons sur la branche next[10] de capstone [23], et pour la manipulation de matrices booléennes creuses sur SciPy [13] et NumPy [19].

La branche next de capstone indique pour chaque instruction le type d'accès sur les registres et opérandes : lecture ou modification. Cela nous permet de trouver rapidement une approximation satisfaisante des dépendances pour la plupart des instructions : tout ce qui est écrit dépend de tout ce qui est lu. Pour pouvoir gérer correctement la pile, les dépendances pour les opérations la concernant doivent être précisées.

Pour le moment notre implémentation ne gère que le x86_64. Nous n'avons dû préciser les dépendances que pour les opérations de type `call`, `ret`, `push`, `pushf`, `pop`, `popf`, ainsi que pour `add rsp, 0x...` et `sub rsp, 0x...`. Toute autre opération sur `rsp` menant à une corruption de pile comme indiqué en section 5.2. Afin d'améliorer la précision nous avons également précisé les dépendances pour `xor` utilisé comme une mise à zéro et `xchg`. Pour toutes les autres instructions du x86_64 et de ses extensions gérée par la branche next de capstone nous utilisons l'heuristique précédemment décrite.

Les représentations matriciels et tabulaires des dépendances sont parfois difficilement lisibles, surtout avec quelques centaines de registres. Nous avons donc inclus un pretty printer qui permet de mettre en évidence tous les registres modifiés, en ignorant les simples décalages de la pile.

[9] `github.com/DGA-MI-SSI/T-Brop`
[10] `https://github.com/aquynh/capstone/tree/next`

```
4889742448     mov qword ptr [rsp + 0x48], rsi
  4883c448     add rsp, 0x48
        5f     pop rdi
      ffe0     jmp rax
```

```
Cost: 38

++++ DepMatrix ++++
rflags <--- rsp
rdi <--- rsi
deref <--- deref, rsp
stack_{-1} <--- rsi

++++ chainCond ++++
rax
```

L'ajout d'une architecture, même exotique, ne devrait pas être trop fastidieux. Pour la plupart des instructions nous n'avons besoin de savoir que si les opérandes sont lues ou modifiées.

8 Évaluation

Nous avons comparé T-Brop en termes de performance à deux outils syntaxiques, rp++ et ROPGadget, et deux outils symboliques, nrop et angrop. Pour ces outils nous avons mesurer le temps nécessaire pour lister les gadgets de fichiers exécutables de taille croissante, sauf pour nrop qui n'active pas son analyse symbolique dans ce cadre, nous avons donc demandé les gadgets équivalents à `mov rax, rbx ; ret`. Nous avons limité la taille des gadgets à 10 instructions ou 22 bytes suivant la limite supportée par les outils. T-Brop obtient des résultats similaire avec ces deux bornes.

Les tests ont été effectué sur une machine virtuelle à deux cœurs cadencés à 2GHz avec 4Go de mémoire RAM. Les résultat obtenus sont présentés dans les tables 2 et 3. Le nombre de gadgets identifiés par les outils est du même ordre de grandeur, on remarque tout de même un facteur deux entre T-Brop et angrop d'une part et rp++ et ROPgadget d'autre part. Cette différence peut s'expliquer par différents facteurs : sections analysées, instructions prises en compte, gestion des doublons... En terme de temps de traitement, comme on pouvait s'y attendre, T-Brop est beaucoup plus lent que les approches purement syntaxiques. En plus du désassemblage vers une chaîne de caractères, T-Brop doit pour chaque instruction identifier les flux d'information, en faire une matrice de dépendance de quelques centaines de dimensions, calculer un produit matriciel, et stocker le résultat. T-Brop reste par ailleurs beaucoup plus rapide que les approches symboliques : pour chaque instruction elles

Binaire analysé		Temps de traitement en minutes:secondes				
Nom	Taille (ko)	rp++	ROPgadget	T-Brop	angrop	nrop
fstab-decode	6	0:00.00	0:00.15	0:00.90	*0:07.66*	0:24.92
java	6	0:00.00	0:00.15	0:00.77	0:07.56	0:33.18
fgconsole	10	0:00.01	0:00.18	0:01.13	0:12.75	1:42.91
nisdomainname	14	0:00.01	0:00.16	0:01.30	0:16.25	3:07.10
openvt	19	0:00.02	0:00.20	0:01.54	0:18.04	5:36.12
echo	31	0:00.06	0:00.30	0:03.67	0:59.79	13:00.73
rmdir	39	0:00.10	0:00.39	0:05.00	*1:11.10*	11:22.58
touch	63	0:00.13	0:00.51	0:07.02	1:38.85	20:11.61
kmod	151	0:00.35	0:01.23	0:18.73	3:59.10	42:00.60
tar	375	0:01.10	0:03.51	0:58.47	*2:45.70*	
c++	898	0:01.81	0:04.86	1:44.97	*15:40.45*	
rbash	1013	0:03.01	0:11.56	2:38.21	*25:53.19*	
static-sh	1918	0:05.74	0:18.22	9:07.77	*47:01.06*	
python3	4360	0:11.27	0:23.45	17:14.36	*21:12.80*	

Tableau 2. Temps nécessaire au traitement de fichiers exécutables de taille croissante. Une cellule vide indique un temps de traitement supérieur à une heure. Une cellule grisée en italique indique un arrêt prématuré dû à une erreur à l'exécution.

Binaire analysé		Nombre de gadgets identifiés			
Nom	Taille (ko)	rp++	ROPgadget	T-Brop	angrop
fstab-decode	6	110	145	72	*Err*
java	6	98	131	61	55
fgconsole	10	188	260	119	110
nisdomainname	14	252	329	159	146
openvt	19	282	391	164	156
echo	31	1209	1218	633	626
rmdir	39	1528	1588	811	*Err*
touch	63	2252	2292	1249	1083
kmod	151	6660	7019	3818	3728
tar	375	20406	24322	10757	*Err*
c++	898	37849	29785	17392	*Err*
rbash	1013	51687	52632	25694	*Err*
static-sh	1918	116147	111916	63384	*Err*
python3	4360	206262	136360	115460	*Err*

Tableau 3. Nombre de gadgets identifiés dans des fichiers exécutables de taille croissante. Une cellule vide indique que l'analyse a été stoppée au bout d'une heure. Une cellule grisée en italique indique un arrêt prématuré dû à une erreur à l'exécution.

doivent faire un calcul symbolique bien plus couteux qu'un calcul matriciel. En plus de ce calcul symbolique nrop requête un solveur SMT pour chaque gadget identifié. Ainsi, sur un binaire de 151ko par exemple, nrop nécessite environ 40 minutes, angrop 4 minutes, T-Brop 20 secondes et rp++ et ROPgadget autour d'une seconde. Précisons que seuls angrop et nrop proposent une implémentation parallèle et profitent de la présence d'un deuxième cœur.

Nous n'avons pas comparé notre approche à PSHAPE, cependant l'auteur rapporte un traitement de 4,7Mo/heure sur un serveur avec 40 cpu Intel Xeon E5 4640 avec 224 Go de RAM [7]. Bien que nous n'ayons pas utilisé les même binaires de test, Il nous semble raisonnable d'affirmer que T-Brop est plus rapide.

Une évaluation de la pertinence de ces différents outils dans le cadre d'une analyse serait souhaitable mais difficile à mettre en place. De façon anecdotique, une analyse avec rp++ dans le cadre du scénario de l'exemple 13 nous a permis d'obtenir un *stack pivot* en cinq gadgets, quand T-Brop nous a proposé en quelques minutes une solution en deux gadgets.

En général les informations renvoyées par T-Brop sont relativement précises malgré les approximations. Reprenons les gadgets de l'exemple 12, et affichons la sortie de T-Brop.

```
    add rbx, [rdx+4*rcx]
    imul rbx
    call [rax+0xC05FEFE]
```

```
++++ DepMatrix ++++
rflags <--- rax, rbx, rcx, rdx, mem_r
rax <--- rax, rbx, rcx, rdx, mem_r
rbx <--- rbx, rcx, rdx, mem_r
rdx <--- rdx, rax, rbx, rcx, mem_r
deref <--- deref, rax, rbx, rcx, rdx, rsp, mem_r

++++ chainCond ++++
rax, rdx, rcx, rbx, mem_r
```

Comme prévu ce gadget induit un grand nombre de dépendances. Lors de nos tests, angrop n'a pas pu analyser ce gadget, ni une version remplaçant le `call` final par un `mov push ret`.

Le second gadget est plus long mais plus simple :

```
    push rax
    push rbx
    push rcx
    mov rcx, rbx
    mov rbx, rax
    pop rax
    pop rdx
    pop rsi
    pop rdi
    ret 0x10
```

```
++++ DepMatrix ++++
rax <--- rcx
rbx <--- rax
rcx <--- rbx
rdi <--- stack_0
rdx <--- rbx
rsi <--- rax
deref <--- deref, rsp
stack_{-7} <--- rcx
stack_{-6} <--- rbx
stack_{-5} <--- rax

++++ chainCond ++++
stack_3
```

Cette fois ci, angrop est plus précis car il précise bien qu'il s'agit d'égalités et non de simples dépendances :

```
Stack change: 0x20
Changed registers: set(['rcx', 'rsi', 'rbx', 'rdx', 'rdi', 'rax'])
Popped registers: set(['rdi'])
Register move: [rbx to rcx, 64 bits]
Register move: [rax to rsi, 64 bits]
Register move: [rax to rbx, 64 bits]
Register move: [rbx to rdx, 64 bits]
Register move: [rcx to rax, 64 bits]
Register dependencies:
    rbx: [rax ()]
    rcx: [rbx ()]
    rdx: [rbx ()]
    rax: [rcx ()]
    rsi: [rax ()]
```

Angrop utilisant une approche symbolique, il est assez facile de construire des exemples ou T-Brop est moins précis. Comme expliqué, T-Brop s'appuie sur capstone, or capstone ne différencie pas les flags. De plus T-Brop approxime les flux en considérant les lectures et écritures indépendamment. Ainsi sur l'exemple suivant :

```
    dec rcx     ; rflags <--- rcx
    lodsq       ; rax = [rsi]
                ; rsi += (DF?-8:8)
```

```
++++ DepMatrix ++++
rflags <--- rcx
rax <--- rcx, rsi, mem_r
rsi <--- rsi, rcx, mem_r
deref <--- deref, rsi, rsp

++++ chainCond ++++
stack_0
```

T-Brop propage une dépendance de `rcx` vers les flags, puis du *direction flag*, lu par `lodsq`, vers `rax`, modifié par cette même instruction. Un meilleur traitement des flags et de `lodsq` permettrait d'obtenir un résultat proche de celui d'angrop :

```
Stack change: 0x8
Changed registers: set(['rcx', 'rsi', 'rax'])
Popped registers: set([])
Register dependencies:
    rcx: [rcx ()]
    rsi: [rsi ()]
Memory read:
    address (64 bits) depends on: ['rsi']
    data (64 bits) stored in regs:['rax']
```

Les alias sont plus problématique. Dans l'exemple suivant T-Brop manque l'écriture sur la pile et la dépendance de `rcx` vers `rax` :

```
    mov     rbx, rsp
    mov     [rbx], rcx
    pop     rax
    ret
```

```
++++ DepMatrix ++++
rax <--- stack_0
rbx <--- rsp
deref <--- deref, rsp
mem_w <--- rcx, rsp

++++ chainCond ++++
stack_1
```

Si angrop semble adopter la même approche que nous pour les alias potentiels, il traite correctement les alias avérés :

```
Stack change: 0x10
Changed registers: set(['rax', 'rbx'])
Popped registers: set([])
Register move: [rcx to rax, 64 bits]
Register dependencies:
    rax: [rcx ()]
```

Il faut cependant garder à l'esprit que dans nos tests, angrop exécuté sur deux cœurs est dix à vingt fois plus lent que T-Brop sur un seul cœur, lorsqu'il termine sans erreur.

9 Travaux futurs

Nous avons plusieurs pistes pour améliorer T-Brop. En terme de temps d'exécution la priorité est une implémentation multi-thread. La recherche de gadget se parallélise naturellement : le binaire analysé n'est accédé qu'en

écriture, et chaque retour de fonction ou branchement indirect peut être traité indépendamment. Angrop propose déjà une telle implémentation.

La précision peut également être améliorée de façon générique en ajoutant par exemple un pseudo-registre pour chaque déréférencement vers une variable globale ([0x...] et [rip+0x...] en x86_64). Nous envisageons également de différencier plusieurs niveau de dépendance pour distinguer les opérations simples comme une affectation ou une addition, des opérations ne concernant que quelque bits ou difficilement inversibles.

Par ailleurs notre fonction de coût doit encore être optimisée pour être plus pertinente. D'autre fonction de coût peuvent également être intéressante, comme GaLity [8], si elles peuvent être calculées rapidement et utilisées en ligne pour guider la recherche de gadgets.

Pour le moment T-Brop n'offre qu'une fonctionnalité de filtrage intelligent sur l'ensemble des gadgets, ainsi qu'une estimation de la taille de la plus petite chaîne permettant d'obtenir une dépendance donnée. Le chaînage automatique est limité aux combinaisons de deux gadgets comme décrit en section 6.2. Pour aller plus loin, nous pouvons voir la matrice de dépendance comme la matrice d'adjacence d'un graphe. On peut donc rechercher des chaînes candidates en cherchant des chemins dans le graphe correspondant à M_G la matrice d'un ensemble de gadgets comme définie section 4.3. Avec cette approche la condition de chaînage ne peut pas être prise en compte.

Une autre approche plus précise consiste à considérer les gadgets comme des transitions dans un graphe dont chaque nœud représente un état du système : pour chaque registre un bit indiquant s'il est influencé par l'analyste ou non. On se retrouve avec un nombre d'états exponentiel en le nombre de registres. Il faut donc réduire le nombre de registres après la phase de création des gadgets et utiliser des algorithmes de plus court chemin pour les très grands graphes. La condition de chaînage peut alors être prise en compte.

10 Conclusion

Nous avons présenté une approche originale pour l'aide à la réalisation de ROP chain basée sur un calcul de matrice de dépendances. Elle permet un filtrage plus expressif que les approches syntaxiques tout en nécessitant des calculs beaucoup moins lourds que les approches symboliques.

L'approche peut être conservative et garantir l'identification de toutes les dépendances au prix d'un certain nombre de faux positifs. Plusieurs

heuristiques relâchant l'exigence de correction ont été présentées pour réduire le nombre de faux positifs.

Nous avons également montré comment utiliser les matrices de dépendances pour évaluer la longueur minimal d'une ROP chain permettant d'obtenir certains transferts d'information. Aucune autre méthode connue ne permet cette fonctionnalité aussi efficacement.

L'approche a été implémentée dans un outil libre prometteur, bien plus rapide que les outils symboliques.

Références

1. Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow Integrity Principles, Implementations, and Applications. *ACM Trans. Inf. Syst. Secur.*, 13(1) :4 :1–4 :40, November 2009.
2. Clark Barrett and Cesare Tinelli. Satisfiability Modulo Theories. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*. Springer, 2018.
3. Julien Couvy. Exploiting ROP attacks with a unique instruction. Bachelor's thesis, Vrije Universiteit Amsterdam, 2017.
4. Christopher Domas. M/o/Vfuscator, the single instruction compiler – Turning 'mov' into a soul-crushing RE nightmare. In *REcon*, 2015.
5. Thomas Dullien, Tim Kornau, and Ralf-Philipp Weinmann. A Framework for Automated Architecture-Independent Gadget Search. In Charlie Miller and Hovav Shacham, editors, *4th USENIX Workshop on Offensive Technologies, WOOT '10, Washington, D.C., USA, August 9, 2010*. USENIX Association, 2010.
6. Nicolás Alejandro Economou. Agafi (Advanced Gadget Finder). In *Ekoparty*, 2014.
7. Andreas Follner. *On Generating Gadget Chains for Return-Oriented Programming*. PhD thesis, Technische Universität Darmstadt, Darmstadt, 2017.
8. Andreas Follner, Alexandre Bartel, and Eric Bodden. Analyzing the Gadgets. In Juan Caballero, Eric Bodden, and Elias Athanasopoulos, editors, *Engineering Secure Software and Systems*, pages 155–172, Cham, 2016. Springer International Publishing.
9. Olivia Lucca Fraser, Nur Zincir-Heywood, Malcolm Heywood, and John T. Jacobs. Return-oriented Programme Evolution with ROPER : A Proof of Concept. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '17, pages 1447–1454, New York, NY, USA, 2017. ACM.
10. Daniel Hedin and Andrei Sabelfeld. A Perspective on Information-Flow Control. In Tobias Nipkow, Orna Grumberg, and Benedikt Hauptmann, editors, *Software Safety and Security - Tools for Analysis and Verification*, volume 33 of *NATO Science for Peace and Security Series - D : Information and Communication Security*, pages 319–347. IOS Press, 2012.
11. Christian Heitman. Compilador ROP. In *Ekoparty*, 2013.
12. Christian Heitman. BARFing Gadgets. In *Ekoparty*, 2014.
13. Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy : Open source scientific tools for Python, 2001–. [Online ; accessed 2018-04-07].
14. Sebastian Krahmer. Der Hammer : x86-64 und das Um-schiffen des NX Bits. In *22nd Chaos Communication Congress – Private Investigations*, 2005.

15. Elias Levy (Aleph One). Smashing the Stack for Fun and Profit. *Phrack*, 7(49), November 1996.
16. Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In Defense of Soundiness : A Manifesto. *Commun. ACM*, 58(2) :44–46, January 2015.
17. Nergal. The advanced return-into-lib(c) exploits : PaX case study. *Phrack*, 0x0b(0x3a), December 2001.
18. Đurica Nikolić and Fausto Spoto. Reachability Analysis of Program Variables. *ACM Trans. Program. Lang. Syst.*, 35(4) :14 :1–14 :68, January 2014.
19. Travis E. Oliphant. *Guide to NumPy*. CreateSpace Independent Publishing Platform, USA, 2nd edition, 2015.
20. Tavis Ormandy. Did OptiROP ever get released ? I always solve these constrained ROP problems manually because no good tools exist. `https://twitter.com/taviso/status/733740666920951808`.
21. Tavis Ormandy (@taviso). I know about DEPLIB, but it's not really usable. I think I'm just going to have to write the tool I want :(/cc @4dgifts. `https://twitter.com/taviso/status/733741022795042816`, May 2016.
22. Nguyen Anh Quynh. OptiROP : hunting for ROP gadgets in style. In *Black Hat USA*, 2013.
23. Nguyen Anh Quynh. Capstone : Next Generation Disassembly Framework. In *Black Hat USA*, 2014.
24. Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-Oriented Programming : Systems, Languages, and Applications. *ACM Trans. Inf. Syst. Secur.*, 15(1) :2 :1–2 :34, March 2012.
25. Rolf Rolles. Synesthesia : Automated Generation of Encoding-Restricted Machine Code. In *Ekoparty*, 2016.
26. Daniel Schoepe, Musard Balliu, Benjamin C. Pierce, and Andrei Sabelfeld. Explicit Secrecy : A Policy for Taint Tracking. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 15–30, March 2016.
27. Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 317–331, Washington, DC, USA, 2010. IEEE Computer Society.
28. Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. Q : Exploit Hardening Made Easy. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 25–25, Berkeley, CA, USA, 2011. USENIX Association.
29. Hovav Shacham. The Geometry of Innocent Flesh on the Bone : Return-into-libc Without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.
30. Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK : (State of) The Art of War : Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, May 2016.
31. Pablo Solé. Hanging on a ROPe. In *Ekoparty*, 2010.

A Décidabilité/Complexité du calcul de matrice de dépendance

Le calcul de la matrice M est indécidable dans le cas général. Pour s'en convaincre, prenons un programme P, ajoutons un registre w n'apparaissant pas dans P, l'entrée correspondant à l'influence de w en entrée sur w en sortie dans la matrice M est égale à 1 si et seulement si P termine. En effet, par construction w n'influence pas l'exécution de P, et les calculs effectués n'influencent pas w. S'il existe une trace finie à partir d'un état initial I, il est donc possible de construite un état I' identique à I sauf sur le registre w tel que I et I' mènent en un temps fini à deux états pour lesquels la valeur de w est différente, et $M(w, w) = 1$.

Réciproquement, si P ne termine pas, alors aucune trace ne mène à un état final, il est donc impossible de construire deux états initiaux menant à deux états finaux pour lesquels la valeur de w est différente, dans ce cas $M(w, w)$ est nul, mais également toutes les autres entrées de la matrice M. Attention, cela ne veut pas dire qu'une matrice de dépendance nulle implique la non-terminaison, par exemple un programme qui initialise tous les registres à une valeur fixe aura une matrice de dépendance nulle.

Ici, nous ne nous intéressons qu'à des séquences d'instructions sans structure de contrôle (et sans réécriture de code), le problème devient alors NP-complet. Sans structure de contrôle le nombre de registres et d'accès mémoires est borné linéairement par la taille du programme. Chaque bit manipulé à chaque instruction de la séquence peut donc être exprimé sous la forme d'une formule logique de taille polynomiale[11] en la longueur du programme. Pour tous registres i et j, $M(i, j)$ peut alors également être exprimé comme un problème de satisfiabilité (SAT) d'une formule logique de taille polynomiale en la longueur du programme : est-ce qu'il existe deux états identiques partout sauf sur le registre j tels que l'application de la séquence d'instruction s mène à deux états pour lesquels le registre i est différent ?

Réciproquement, tout problème SAT peut se réduire à un calcul de matrice de dépendance. Étant donnée une formule SAT S et une séquence s permettant de l'évaluer, la ligne dans M_s correspondant au résultat de cette évaluation est nulle si et seulement si S est une tautologie ou une antinomie. Une simple évaluation de S permet de distinguer les deux. Ainsi, le calcul de matrice de dépendance et SAT appartiennent à la même classe de complexité.

[11] Polynomiale et non linéaire car pour chaque accès mémoire il faut vérifier si l'adresse déréférencée correspond à un accès mémoire précédent.