École des Mines de Nancy
Département Information et Système
Rapport de projet 2A

Encadrant : Jean-Yves MARION

Analyse de codes exécutables

Philippe BECHTEL
Hubert GODFROY
Jean MILLOT

Nancy, le 1er août 2012

Table des matières

1 État de l'art **4**
1.1 Fonctionnement d'un désassembleur. 5
1.1.1 Que doit faire un désassembleur ? 5
1.1.2 Différentes manières d'effectuer le désassemblage. 6
1.1.3 Quels sont les problèmes rencontrés ? 7
1.2 Techniques d'obfuscations 7
1.2.1 Data flow obfuscation 8
1.2.2 Control flow obfuscation 9
1.2.3 Différences entre data flow Obfuscation et control flow Obfuscation . . . 10
1.3 Exemple d'obfuscateur 12
1.3.1 Présentation de l'obfuscateur utilisé 12
1.3.2 Présentation des résultats 12

2 Conventions du désassemblage **13**
2.1 Construction du graphe de flow 14
2.2 Méthode de représentation 16
2.2.1 Dans le programme 16
2.2.2 Représentation persistante 16
2.2.3 Application : détection des *junk bytes* 16
2.3 Propagation des constantes 17
2.3.1 Adaptation de la méthode 17
2.3.2 Allègement du graphe 18
2.3.3 Détermination des sauts indéfinis 18

3 Utilisation du désassembleur **18**
3.1 Organisation des fichiers sources 19
3.1.1 Contenu des fichiers 19
3.1.2 Dépendances 19
3.2 Machine virtuelle 19
3.2.1 Classe 19
3.2.2 Registres 19
3.2.3 Mémoire 20
3.2.4 Variable 21
3.2.5 Pile d'appel 22
3.2.6 Processeur 22
3.3 Implémentation des instructions 23
3.3.1 Mode de représentation 23
3.3.2 Ajout d'une instruction 23
3.4 Fonctions utilisateur 25
3.4.1 L'objet `Desas` 25
3.4.2 Ouverture d'un fichier exécutable 25
3.4.3 Construction du graphe de flow 26

3.4.4 Traitement du graphe 26
3.5 Portabilité 27

Annexes **27**

Introduction

Parmi la multitude de problèmes que l'on rencontre avec le développement de l'informatique à tous les niveaux ; il y a ceux qui trouvent leurs solutions dans l'obfuscation et d'autre part, il y a ceux où c'est l'obfuscation de programme qui est à l'origine du problème. Et ces problèmes ne sont pas de moindre importance :

- **La propriété intellectuelle** ou, comment s'assurer que les logiciels que l'on distribue ne soient pas compréhensibles par les concurrents.
- **La lutte contre la contrefaçon** à l'aide de bouts de codes à la fois essentielle au fonctionnement et identifiant le logiciel tout en étant difficile à localiser dans l'ensemble du code.
- **La lutte contre les malwares** qui se complexifient et deviennent de plus en plus difficiles à détecter avec l'arrivée de virus métamorphiques.

Autant de champs d'applications qui ont de près ou de loin pour sujet d'études l'obfuscation. C'est pour cela que dans ce projet de deuxième année nous allons essayer de ramener l'analyse d'un exécutable à celle d'une représentation normalisée pouvant s'affranchir des problèmes soulevés par l'obfuscation. De plus, pour connaitre l'effet d'un programme sans en effectuer l'exécution [1] il faut connaitre l'état de l'ordinateur à chaque étape de la suite d'instructions. C'est pour cela que l'on va essayer également de simuler cette exécution. Nous avons pour cela besoin dans un premier temps de comprendre le fonctionnement des techniques d'obfuscations et les difficultés inhérentes à la décompilation. Une fois que l'on se sera doté d'outils pour analyser les codes machines, nous pourrons travailler sur la représentation logique du fonctionnement d'un programme. L'ensemble des informations disponibles par décompilation va aussi nous permettre d'étudier ses effets mais aussi de résoudre des problèmes liés à l'obfuscation.

1 État de l'art

L'assembleur joue un rôle central dans les processus de compilation. Il est alors primordial lorsque l'on souhaite décompiler un code (reverse engineering) de s'assurer que l'on est au moins capable de le désassembler : c'est à dire, récupérer un code assembleur correspondant au code machine. Bien que l'assembleur soit en théorie en bijection avec le code machine, il y a un manque d'informations logique pour que cela soit fait de façon sûre et immédiate [2]. Nous allons donc dans cette partie voir le fonctionnement d'un désassembleur et identifier quelles sont les difficultés rencontrées lors de la conception d'un désassembleur. Dans un deuxième temps, nous allons mettre en évidence certaines méthodes d'obfuscations de niveau assembleur qui reposent sur les particularités de ces langages et des différents types de désassembleurs.

1. pour pouvoir effectuer l'analyse sur des programmes malveillants par exemple
2. Il manque les commentaires, les noms des variables par exemple.

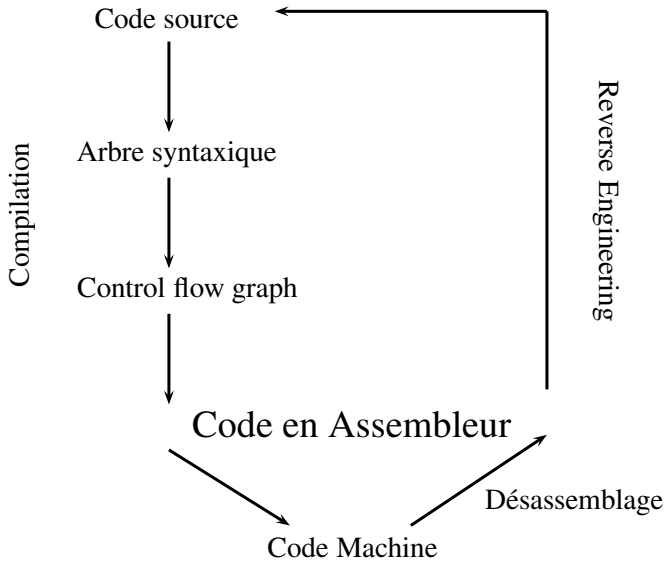


FIGURE 1 – La place de l'assembleur dans le cycle Compilation/Décompilation

1.1 Fonctionnement d'un désassembleur.

1.1.1 Que doit faire un désassembleur ?

Le désassembleur doit pouvoir, une fois qu'il connaît le set d'instructions utilisé par le processeur et le point d'entrée du fichier exécutable, décrypter le code machine pour donner son équivalent en un langage assembleur. C'est à dire, le programme que l'on doit obtenir en recompilant le code machine désassemblé doit avoir exactement le même comportement.
Pour cela il faut d'abord récupérer le point d'entrée du programme, c'est-à-dire l'endroit où, une fois que le fichier exécutable a été chargé en mémoire, le contrôle est cédé par l'OS au programme qui va s'exécuter. Pour cela, il est nécessaire de regarder les en-têtes (header) des différents types de fichiers exécutables (Ils sont plus ou moins conçus de la même manière et ceux qui sont principalement utilisés sont les formats Mach-o, ELF et PE).
Une fois le point d'entrée extrait, on peut démarrer le désassemblage. Cela consiste à identifier :

- les opcodes : c'est la partie d'une instruction en langage machine qui définit l'opération à effectuer.
- la taille de l'instruction (jusqu'où on considère que les bits constituent une instruction)
- les arguments

On peut en déduire, connaissant le set d'instructions utilisé par le processeur la taille complète de l'instruction et des arguments, et donc reconstituer la ligne assembleur correspondant à cette opération. Le désassembleur agit donc comme un dictionnaire entre le langage machine et l'assembleur, compréhensible par les humains. Une fois l'opération décodée, il faut continuer le désassemblage jusqu'à arriver au `ret` (return) final. Cependant, il y a plusieurs façons de faire pour suivre le déroulement d'un programme et atteindre la fin.

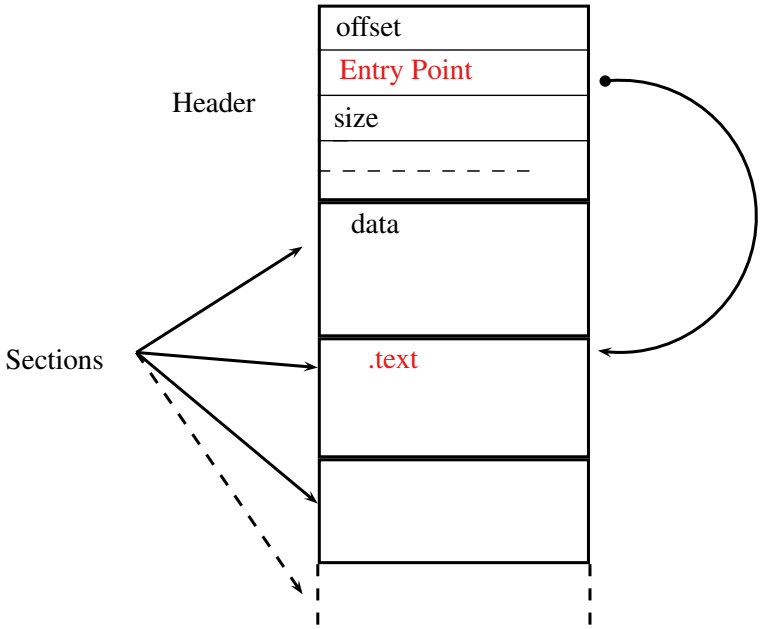


FIGURE 2 – Structure d'un fichier exécutable

1.1.2 Différentes manières d'effectuer le désassemblage.

Afin de désassembler un programme il y a deux manières de procéder : statiquement et dynamiquement. Dynamiquement signifie que l'on exécute le programme et qu'on analyse chaque action du programme (un peu comme ce que fait un debbuger).

Statiquement, plusieurs façons sont possibles :

- **Balayage linéaire :** il effectue la traduction de toutes les instructions de la section `.text` depuis le point d'entrée, les unes après les autres. Cela correspond au premier programme présenté dans la partie BeaEngine. C'est ce type de désassemblage qui est le plus utilisé (l'outil Objdump de GNU par exemple).

- **Balayage récursif transversal :** On procède de la même manière que précédemment mais au lieu de continuer systématiquement à décoder, on essaye de reconnaitre les appels de fonctions, les `jump` et de poursuivre le désassemblage à l'adresse ciblée par ces instructions.

Nous n'avons travaillé qu'avec les désassembleurs statiques et c'est pour cela que l'on va décrire plus en détails leurs spécificités (ou défauts).

1.1.3 Quels sont les problèmes rencontrés ?

Bien que l'on n'ait pas encore mis en place de désassembleur dynamique on s'aperçoit d'ors et déjà que l'on va rencontrer un problème singulier : l'intégralité du programme ne sera pas traduit. En effet seul le code appelé durant l'exécution précise va être parcouru (donc désassemblé). Or l'origine des paramètres pris en compte par le programme à désassembler est totalement inconnue. Ces paramètres peuvent être les valeurs prises par certains registres, sur la pile d'appels ou encore situées à un emplacement mémoire précis. Il est donc difficile de trouver les paramètres nécessaires à la décompilation totale du programme. De plus, cette façon de procéder peut-être sensiblement plus longue (souvent de plusieurs facteurs) car il est très courant que lors de l'exécution, des bouts de code soient exécutés plusieurs fois.

Les désassembleurs statiques ne sont pas dépourvus de problème d'implémentation, bien au contraire.
Le balayage linéaire n'interprète jamais ce qu'il lit et peut par conséquent tomber sur des données dans la section `.text` sans savoir que ce ne sont pas des instructions et tenter de les traduire. Cela va donner des interprétations fausses, sauf si le désassembleur ne peut pas trouver d'opcodes équivalents aux bits machines lues : on aura dans ce cas une erreur. De plus, cette technique ne désassemble pas non plus l'intégralité du programme puisque celui-ci peut à l'aide d'instructions comme `jump Addr` aller exécuter une partie de code n'étant pas comprise au sein de la partie `.text`.

C'est dans l'optique de corriger ce dernier défaut que l'on préfère utiliser le balayage récursif transversal.
Cependant, l'implémentation d'un désassembleur de ce type soulève un nouveau problème : est-on capable de déterminer l'adresse de destination de n'importe quel type d'appels ? En effet, si lors du processus de désassemblage on tombe sur une instruction comme `jump eax` (où `eax` est un registre) ; il faut être capable de connaitre le contenu de `eax` pour pouvoir continuer le désassemblage à l'emplacement désigné.

Ce sont les lacunes de chacun des désassembleurs que l'on va pouvoir utiliser pour obfusquer le code et rendre le résultat du désassemblage complètement erroné.
Nous avons utilisé les articles [**?**] et [**?**] pour cette partie.

1.2 Techniques d'obfuscations

Nous avons pu voir dans la partie précédente quels étaient les enjeux et les limites d'une méthode de désassemblage de qualité. On appelle techniques d'obfuscation les procédés dont le but est de rendre aussi inintelligible que possible le code binaire d'un exécutable. Nous allons dans cette partie étudier, au regard des limites des méthodes de désassemblage, les différentes méthodes qui permettent d'empêcher la bonne traduction du code binaire d'un exécutable en langage assembleur. Ces méthodes peuvent par exemple servir à la construction de virus capables d'obfusquer eux même leur propre code, ils sont appelés virus métamorphiques. Nous distinguerons dans cette partie 2 types d'obfuscation : la première engloble les

méthodes qui s'attachent à modifier les données du code d'un fichier (Data Flow Control). La deuxième concerne les méthodes permettant de jouer sur l'ordre d'enchainement des différentes instructions (Control Flow Obfuscation).

1.2.1 Data flow obfuscation

Instruction substitution

Cette méthode consiste simplement à substituer un bloc d'instructions par un autre sans pour autant modifier le comportement du programme. Cette méthode se base sur le fait qu'il est possible d'écrire une multitude de programmes différents effectuant la même opération.

Listing 1 – Exemple de substitution d'instructions

```
int i = 0 ;
For(int k = 0 ; k < 10 ;k++){
i = i +k ;
}
```

Peut aussi s'écrire :

```
int i = 0 ;
int k = 0 ;
while(k < 10){
i = i +k ;
k++
}
```

Il est clair que les deux portions de code présentées effectuent la même opération, cependant elles ne sont pas représentées par le même code.

Permutation d'instructions

Cette méthode consiste simplement à échanger 2 blocs d'instructions tout en prenant garde de ne pas modifier le comportement du programme. Exemple :

- si on prend l'exemple d'une multiplication, il est clair que l'on peut inverser les lignes `a = a * 2;` et `a = a*4;` sans pour autant modifier l'état final de a.
- On ne peut par contre évidemment par inverser les lignes `a = a + 1 ;` et `a = a*2;` .

Substitution de variables et insertion de dead code

La méthode élémentaire de substitution des variables consiste tout simplement à changer le nom des variables dans un programme. Cela permet notamment d'enlever tous les noms usuels ou cohérents des différentes variables.

Un dead code est un enchainement d'instructions dont le résultat est nul. On peut donc l'insérer à presque tout endroit du code sans que cela change le résultat du programme. Un

exemple évident de dead code serait `a = a` ou encore `a = a +1 ; a = a - 1 ;`. Bien que ces exemples soient élémentaires, il existe des éxemples bien plus complexes dans lesquels il est difficile de déterminer si les instructions sont ou ne sont pas inutiles.

Insertion de junk bytes

Comme nous le verrons cette méthode de désassemblage est principalement utilisée pour contrecarrer un désassemblage linéaire du byte code. Lors du passage du code assembleur vers le code binaire d'un programme, chaque instruction assembleur est représentée par un certain nombre de bytes. L'insertion de junk bytes consiste à ajouter dans le fichier binaire un certain nombre de bytes représentant une instruction assembleur non complète. On crée ainsi un décalage qui empêche la bonne traduction en assembleur. Cependant, il faut s'assurer que ces portions d'instructions ne soient jamais atteintes lors de l'exécution de programme. Ainsi il est impératif que l'instruction précédent directement un junk bytes soit un saut inconditionnel.

1.2.2 Control flow obfuscation

Le but principal de l'obfuscation par modification du control flow consiste à diviser un programme en un certains nombres de blocs d'instructions et de faire un sorte que l'exécution du programme ne soit pas linéaire en faisant appel à des sauts conditionnels ou inconditionnels ou encore à des appels de fonctions.

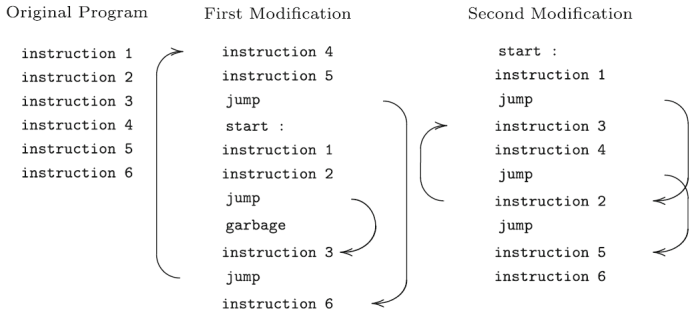


FIGURE 3 – Illustration de l'altération du Control flow

Jump Table

Le but principal de l'utilisation des sauts est de faire en sorte que le désassembleur ne sache pas suivre le chemin d'un saut. Pour cela, on utilise très généralement des sauts conditionnels dont la réalisation dépend de l'état d'un flag lors de l'exécution. On divise ainsi le programme en une succession de blocs d'instructions se terminant tous par un saut conditionnel. On construit alors une Jump Table qui détermine l'état des variables booléennes qui définissent

quels sauts doivent être suivis lors de l'exécution de manière à conserver le même enchainement des instructions bien qu'un saut inconditionnel soit fait à la fin de chaque bloc.

Imaginons que l'on sépare le programme en k blocs. On construit alors k variables booléennes Si tel que pour tout i , si K_i = `True`, alors le saut à la fin du $i$ème bloc d'instructions est emprunté. Si K_i = `False` on prend la branche par défaut qui va envoyer le désassembleur vers un bloc qui n'est pas celui emprunté normalement.

De la même manière, on peut utiliser les propriétés d'une suite pour calculer les valeurs prises par le flag. On peut imaginer que K_n est la suite de syracuse ($K_{n+1} = K_{n/2}$ si paire et $3K_n + 1$ sinon). Et incrémenter dans chaque bloc la suite et utiliser la conjecture (qu'au bout d'un certain rang n, pour K_0 choisi on ait le cycle 4,2,1) afin de faire les tests appropriés à la fin de chaque bloc pour que le saut choisi soit le bon. Si lors du désassemblage on ne trouve pas le bon K_0 pour lequel les K_n vont prendre les valeurs attendues, l'ordre de désassemblage va être complètement faux[3].

Ces méthodes ont pour but de rendre le plus obscure possible la destination des sauts de chaque bloc pour une personne qui cherche à désassembler. Cependant l'évolution de K est bien connue de son créateur et va déterminer l'unique chemin valide que le programme doit suivre pour fonctionner.

Opaque predicates

Le but consiste ensuite à rendre aussi incompréhensible que possible le tableau de variables. On utilise alors des Opaque predicates qui permettent de faire en sorte qu'une variable booléenne soit tout le temps vraie ou fausse tout en s'efforçant de rendre aussi difficile que possible la connaissance de son état par un programme de désassemblage.
Exemple : On sait que pour que l'exécution se passe correctement, il faut que la variable Sk soit vraie. On peut alors utiliser les instructions suivantes.

`If(a*(a+1)%2 == 0)` S_k `= true Else` S_k `= false`

Où `a` représente un entier. Il est alors clair que la condition `a*(a+1)%2 == 0` est toujours vraie cependant il est difficile pour un algorithme de désassemblage statique de déterminer l'état de cette condition.

1.2.3 Différences entre data flow Obfuscation et control flow Obfuscation

Niveau d'abstraction

La différence majeure entre ces deux classes de méthodes d'obfuscation est qu'elles ne jouent pas sur le même niveau d'abstraction d'un programme. En effet, on pourrait dire que les

3. Ces suites sont ce qu'on appelle des clefs de désassemblage.

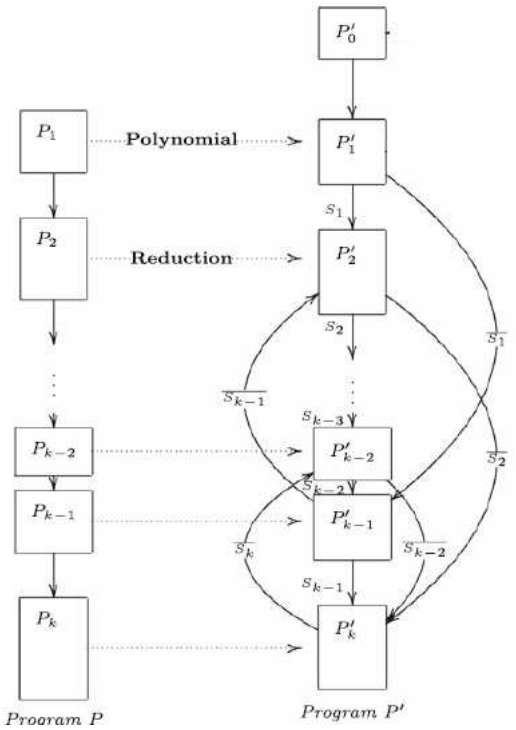


FIGURE 4 – Exemple d'obfuscation à l'aide d'une Jump Table

méthodes de modification de données sont des méthodes superficielles qui ne modifient pas l'essence du programme alors que les méthodes de flow control sont des méthodes qui modifient directement le graphe de flow d'un programme.

Différents impacts sur les méthodes de désassemblage

Nous avons vu en première partie qu'il existait globalement 2 méthodes de désassemblage : la méthode Linear Sweep qui permet de suivre la progression du code de manière linéaire sans prendre en compte les sauts et la méthode Recursive Transerval qui permet de suivre les branches du graphe de flow et de désassembler le programme dans son ensemble en prenant en compte les sauts.
Ainsi la méthode d'insertion de junk instructions est utile pour contrecarrer la méthode de désassamblage Linear Sweep. Les méthodes de changement du flow control sont quant à elles utiles pour contrecarrer le désassemblage Recursive Traversal car son but est d'empêcher les désassembleurs de pouvoir suivre les différents sauts.

1.3 Exemple d'obfuscateur

Dans cette partie nous allons présenter un exemple d'obfuscateur réalisé par l'université d'Arizona et nous baser sur leurs résultats expérimentaux pour montrer l'efficacité des méthodes d'obfuscations.

1.3.1 Présentation de l'obfuscateur utilisé

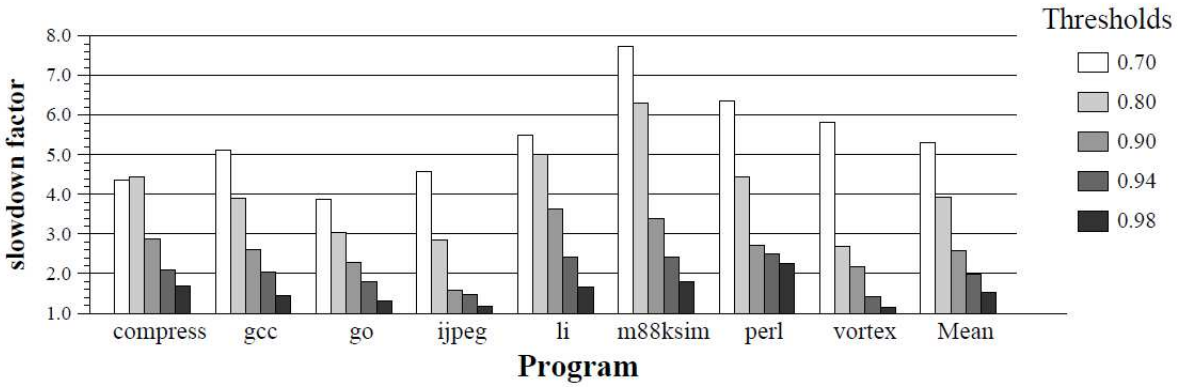
La méthode d'obfuscation utilisée est la suivante : On divise le programme en un certain nombre de blocs. On fait terminer chacun de ces blocs par un saut conditionnel dont la condition est toujours soit vraie soit fausse de manière à conserver l'ordre d'exécution des instructions initial (Utilisation d'opaque predicates).
On remplace ensuite les sauts conditionnels suivant les blocs qui sont exécutés un grand nombre de fois par des appels de fonctions (call en assembleur) ce qui permet une exécution plus rapide du programme. La distinction entre les blocs souvent exécutés (qu'on appelle "hot") et les autres est effectuée à l'aide d'un paramètre θ qui définit le pourcentage des blocs du programme qui seront considérés comme exécutés un grand nombre de fois.
On effectue ensuite la comparaison entre le code assembleur obtenu par le désassemblage et le code assembleur qui correspond au programme initial (sans obfuscations) afin d'obtenir un facteur de confusion (pourcentage des instructions mal traduites).

1.3.2 Présentation des résultats

Temps d'exécution

La première constatation qui peut être faite est que le programme obfusqué a un temps d'exécution plus grand que le programme initial. Cette différence de temps d'exécution dépend du paramètre θ. En effet, θ grandit avec le nombre d'instruction "hot" et donc plus il y a d'appels

de fonctions et donc plus le temps d'exécution est faible. Cela est du au fait qu'un `call` est plus rapide à exécuter qu'un `jne` en assembleur.
Le tableau suivant représente le quotient T1/T0, T1 étant le temps d'exécution du programme obfusqué et T0 celui du programme original. Les tests ont été réalisés sur plusieurs programmes et avec des paramètres θ différents.



(b) Slowdown in execution speed

FIGURE 5 – Ralentissement après obfuscation selon les compilateurs

Taux d'erreur

Le deuxième paramètre qu'il est important d'observer est le taux d'erreur du désassemblage. On distingue le taux d'erreur sur les blocs, le taux d'erreur sur les fonctions et le taux d'erreur sur les instructions. La différence entre ces taux d'erreur est la suivante : On considère qu'un bloc ou une fonction est mal désassemblé si seulement une de ses instructions est mal désassemblée. Le taux de confusion est ensuite donné comme le pourcentage d'élément mal désassemblé. Les tests sont effectués en utilisant les 2 méthodes de désassemblage c'est à dire la méthode récursive et la méthode linéaire.
La première constatation importante à faire est que le pourcentage d'erreur reste très élevé dans les 2 cas. On peut ensuite faire la constatation que la méthode de désassemblage récursive est plus efficace et cela est du au fait comme nous l'avons déjà vu que cette méthode est bien plus robuste aux obfuscations qui jouent sur le graphe de Flow.

2 Conventions du désassemblage

Nous allons présenter ici comment s'effectue l'acquisition des données, et comment les informations sont extraites du fichier exécutable.

PROGRAM	Confusion factor (%)					
	LINEAR SWEEP			RECURSIVE TRAVERSAL		
	Instructions	*Basic blocks*	*Functions*	*Instructions*	*Basic blocks*	*Functions*
compress	82.0	59.6	88.0	47.6	26.7	51.1
gcc	70.5	52.8	86.7	34.1	15.1	50.7
go	75.5	55.7	92.7	38.3	19.2	50.1
ijpeg	75.4	58.2	88.5	39.8	22.4	47.7
li	77.5	57.3	77.1	44.4	24.6	39.8
m88ksim	80.0	59.3	90.0	44.2	24.2	51.1
perl	75.2	56.5	89.0	44.0	24.3	49.9
vortex	65.4	50.6	92.9	34.5	17.8	43.4
Geo. mean	75.0	56.2	88.0	40.6	21.4	47.8

Figure 7: Efficacy of obfuscation: confusion factors ($\theta = 0.98$)

2.1 Construction du graphe de flow

Un désassemblage est défini par 3 paramètres :

- L'adresse mémoire du point d'entrée de l'exécutable chargé en mémoire
- L'adresse virtuelle de ce point d'entrée (c'est à dire l'adresse affichée par le désassembleur)
- La taille du bloc à désassembler.

Après lecture de ces données (cf annexes), le désassembleur lit les instructions à partir de la première et suit toutes les branches de manière récursive. Ainsi seules les branches atteignables par le programme sont désassemblées [4]. Le graphe de flow peut alors être extrait.
Il est intéressant de noter que lors du désassemblage on essaie autant que possible d'éviter les erreurs : en effet, toutes les instructions ne sont pas compréhensibles à première lecture. Par exemple, un `jmp rax` ne peut être compris à ce stade de l'étude car on ignore la valeur contenue dans le registre `rax`. Afin de désassembler le maximum d'instructions possible, tout en s'arrangeant pour que le désassemblage garde un sens, nous avons adopté des conventions d'arrêt ou de passage. Cependant, toutes les erreurs sont retenues dans le graphe. Dans de telles conditions, le désassemblage global ne stoppe jamais, mais l'étude d'une branche s'arrête prématurément si l'instruction suivante est indéterminée. L'ensemble des types d'erreurs est enregistré dans une énumération (voir le listing 2) décrivant l'état de chaque nœud du graphe [5].

4. On verra par la suite que cette méthode nous permettra de détecter les états inaccessibles, et surtout les *junk bytes*, qui sont des instructions dont le début commence au milieu d'une autre.

5. Elle contient également deux autres constantes permettant d'annoncer qu'un nœud a plusieurs fils ou pas

Listing 2 – états possibles d'un nœud

```
enum ValeurEtat{
    SANS_INTERET,
    GO_AND_LEAVE,
    OPCODE_INCONNU,
    DEPASSEMENT_BLOC,
    SAUT_INCOND_OUT_OF_BLOCK,
    SAUT_INCOND_INDEFINI,
    FIN_BLOC_SANS_POINT_ARRET,
    CALL_TERMINAL_OOB,
    CALL_TERMINAL_INDEFINI,
    CALL_FIN_BLOC,
    CALL_INDEFINI,
    CALL_OUT_OF_BLOCK,
    SAUT_COND_FIN_BLOC,
    SAUT_COND_INDEFINI,
    SAUT_COND_OUT_OF_BLOCK,
    SAUT_COND_TERMINAL,
};
```

Les sauts hors du bloc sont souvent dus à des appels de fonctions dynamiquement liées. Ces cas de passage sont reperés par les constantes

- `CALL_OUT_OF_BLOCK`,
- `CALL_TERMINAL_OOB`,
- `SAUT_INCOND_OUT_OF_BLOCK`,
- `SAUT_COND_OUT_OF_BLOCK`.

Les erreurs de lecture sont dues à un problème lors du désassemblage d'une instruction. Elles peuvent être provoquées par la lecture d'un opcode inconnu (`OPCODE_INCONNU`) ou d'un dépassement du bloc lors de la lecture de cette instruction(`DEPASSEMENT_BLOC`).

Les sauts indéfinis sont provoqués par l'utilisation de la valeur d'un registre ou d'un emplacement mémoire comme adresse de saut (par exemple `jmp rax`). Ils sont repérés par les constantes

- `SAUT_INCOND_INDEFINI`,
- `SAUT_COND_INDEFINI`,
- `CALL_INDEFINI`,
- `CALL_TERMINAL_INDEFINI`.

Les opérations en fin de blocs différentes d'un `ret`, d'un `jmp` ou d'un `hlt` lèvent une erreur car il existe une branche dont le chemin est indéterminé. Les constantes utilisées pour ce type d'erreur sont

- `CALL_TERMINAL_OOB`,

- `CALL_TERMINAL_INDEFINI`,
- `CALL_FIN_BLOC`,
- `SAUT_COND_FIN_BLOC`.

2.2 Méthode de représentation

2.2.1 Dans le programme

Lors du chargement du programme en mémoire, à chaque octet est associé un nœud. D'autre part, chaque instruction est représentée par le nœud associé à son premier octet. Les autres nœuds sont alors marqués comme recouverts [6]. Puis, lors du désassemblage en suivant les branchements, ces nœuds sont reliés entre eux pour former le graphe.

2.2.2 Représentation persistante

Nous utilisons la structure des fichiers `.dot` couplé au logiciel `graphviz` pour avoir une représentation graphique de graphe que nous avons construit. On s'interresse ici à la structure du graphe et pas à une étude qualitative de chaque nœud. C'est pourquoi le graphe obtenu dans l'étape de construction est traité au préalable par une fonction de simplification qui supprime les nœuds ayant un unique fils et étant lui-même fils d'un nœud à fils unique.
La figure **??** page **??** donne le graphe de flow pour le programme `fibo` (listing **??**) : Les nœuds rouges représentent les `call`, les bleus les `jmp`, les verts les sauts conditionnels et les gris les `ret` et `hlt`. Les flèches rouges désignent la nouvelle fonction appelée et les vertes le chemin si le saut est valide.
La construction du graphe de flow est une première étape dans le travail de désassemblage au sens large. L'étude va porter à présent sur la sémantique des instructions en s'appuyant sur ce graphe. Le but sera de mettre en évidence le maximum d'informations possible, soit en simplifiant le graphe, soit en l'étendant aux branches indéfinies en déterminant les sauts.

2.2.3 Application : détection des *junk bytes*

Un *junk byte* est le recouvrement d'une instruction par une autre : une instruction s'écrit en général sur plus d'un octet. Si tel est le cas, un saut permet très facilement de placer le registre IP entre deux instructions, adresse normalement inaccessible en cas de lecture linéaire. La figure 6 donne une illustration d'une telle manipulation.

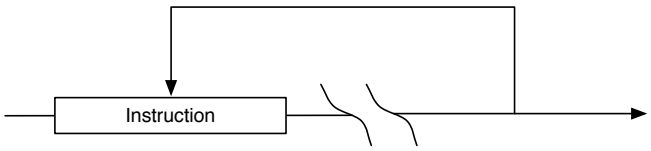
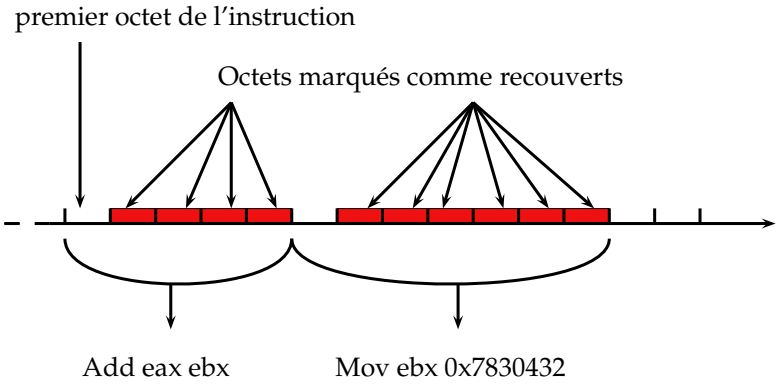


FIGURE 6 – Principe du *junk byte*

6. Pour économiser de la place, chacun de ses nœuds pointe vers le même nœud

La détection des *junk bytes* est permise par la représentation de chaque instruction par le nœud associé à son premier octet et par le marquage des nœuds suivants comme "recouverts". En effet, s'il existe un nœud du graphe qui possède la qualité d'être recouvert, ce nœud est par conséquent issue d'un *junk byte*. Cette méthode nous permet de localiser tous les *junk bytes* issus de la première passe de désassemblage.



2.3 Propagation des constantes

Cette méthode a pour but d'expliciter les constantes pour chaque nœud du graphe. Ayant une meilleure connaissance de ces constantes, l'interprétation du graphe sera plus facile.

2.3.1 Adaptation de la méthode

L'algorithme donné par KILDALL [**?**] utilise une structure de liste afin de comptabiliser l'ensemble des nœuds dont les constantes doivent être mises à jour. Nous avons préféré utiliser le programme récursif suivant en pseudo-code

Listing 3 – propagation des variables

```
1  static void optimizePool_aux(Noeud* n, Noeud* pere){
2          Pool* copyPool = newPoolCopy(pere->pool);
3          f(pere, copyPool);
4          // f modifie copyPool et y stock le nouveau pool
5          int inc = incluDans(n->pool, copyPool);
6          if (inc != NON_INCLUS){
7                  delete(copyPool);
8                  return;
9          }
10         inter(n->pool, copyPool); // intercection dans n->pool
11         delete(copyPool);
12         List* list = n->listeFils;
13         if (list == NULL) {
14                 return;
15         }
16         for (Noeud* N in list) {
17                 optimizePool_aux(N, n);
```

```
18          }
19  }
20  void optimizePool(Noeud* n, const Processeur* initialPool){
21          int inc = incluDans(g->pool, initialPool);
22          if (inc != NON_INCLUS){
23                  return;
24          }
25          inter(n->pool, initialPool); // intercection dans n->pool
26          if (n->listeFils == NULL) {
27                  return;
28          }
29          List* list = n->listeFils;
30          for (Noeud* N in list) {
31                  optimizePool_aux(N, n);
32          }
33  }
```

Il est relativement simple de constater que cet algorithme est équivalent à celui donné précédemment, excepté l'appel récursif des lignes 17 et 31. En effet, cet appel pouvant modifier le pool n, l'ensemble des appels sur les fils de n n'est pas consistant. Cependant, on démontrera en annexe (page 30) que le résultat est inchangé.
Cette implémentation a plusieurs avantages dont celui d'optimiser l'utilisation mémoire. En effet, chaque pool peut prendre beaucoup de place en mémoire (de l'ordre du Ko pour un seul nœud). Alors que l'algorithme initial demande la création d'un nouveau pool pour chaque appel récursif, notre algorithme se déroule en taille mémoire constante.

2.3.2 Allègement du graphe

L'algorithme de propagation des constantes permet de connaître l'état des registres à chaque nœud, en particulier pour le registre de flags. Ce registre déterminant le déroulement du programme, la connaissance d'un drapeau pourra éventuellement permettre de supprimer des branches du graphe original. Si tel est le cas, on appliquera une nouvelle fois l'algorithme de propagation des constantes pour tenir compte de ce "débranchement".

2.3.3 Détermination des sauts indéfinis

De même, il sera éventuellement possible de créer de nouvelles branches si des sauts dont l'adresse cible est *a priori* inconnue sont déterminés par la propagation des constantes. **Cependant**, il convient de remarquer que cette nouvelle branche peut diminuer la détermination du graphe. En effet, si elle rejoint un nœud du graphe déjà existant, elle peut induire des modifications sur les variables déterminisées avant branchement.

3 Utilisation du désassembleur

Nous avons essayé d'implémenter le désassembleur d'une manière la plus modulaire possible. Bien que notre désassembleur ne prenne en compte pour l'instant que peu d'instruc-

tions assembleur, nous avons fait en sorte que l'implémentation future de nouvelles instructions soit simple. D'autre part nous avons essayé de rendre son utilisation assez souple (chargement d'un programme et analyse pour Mach-o et ELF automatique) et surtout transparente. C'est pour cela que le code est très commenté.

3.1 Organisation des fichiers sources

3.1.1 Contenu des fichiers

Chaque fichier `.c` autre que `main.c` et `test.c` est accompagné d'un fichier header donnant le prototype des fonctions réutilisables dans les autres fichiers. Il y a également quelques fichiers header supplémentaires comprenant

- l'ensemble des constantes du programme (`macro.h`)
- des macros permettant d'accéder à la saisie de code uniquement (et donc *a priori* inutile après compilation de la bibliothèque) (dans le fichier `_macro_Build.h`). Ces macros permettent entre autres d'accéder facilement au registre d'un processeur simplement en invoquant son nom (par exemple invoquer le registre eax par `_EAX`) pourvu que le processeur s'appelle `proc`.

3.1.2 Dépendances

Le projet ne dépend que des bibliothèques `Beaengine` et `libelf` pour son utilisation. Les classes de test requièrent en plus la bibliothèque C-unit.

3.2 Machine virtuelle

Afin d'avoir le maximum d'informations sur les effets du programme sans pour autant l'exécuter, nous essayons de faire virtuellement toutes les opérations faites par un programme sur un processeur simulé.

3.2.1 Classe

En accord avec l'algorithme de KILDALL, chaque variable doit être associée à une classe. Dans le simple cas de la propagation des constantes, les deux classes sont `CLASSE_DEFINIE` et `CLASSE_NON_DEFINIE`. C'est pourquoi chaque variable définie par la suite devra se rapporter à une classe.

3.2.2 Registres

La difficulté pour les registres a été de pouvoir gérer l'appel de n'importe quel sous-registre, tout en garantissant que la modification d'un registre d'une famille (par exemple la famille `RAX`, `EAX`, `AX`, `AH`, `AL`) modifie de manière adéquate les autres registres de la famille. On voulait également que l'appel à un registre soit simple et puisse se faire directement. Pour cela, nous avons choisi (comme on peut le voir avec les registres filsl(ow) et filsh(igh) dans la structure)

de représenter les registres par un arbre binaire. On peut donc adapter la taille des registres et le nombre et/ou la taille de ses subdivisions.

Listing 4 – Structure de registre

```
typedef struct _registre{ // 32 bytes
    int                 taille;
    uint64_t            valeur;
    int                 classe;
    struct _registre*   filsl;
    struct _registre*   filsh;
}Registre;
```

Les fonctions principales permettant de gérer les registres sont les suivantes :

- `uint64_t getRegVal(const Registre* reg)` permet d'accéder à la valeur d'un registre et ce quelle que soit la classe de ce registre.
- `uint64_t setRegVal(Registre* reg, uint64_t n)` permet d'affecter une valeur à un registre. Si le registre ou l'un de ses fils est de classe `CLASSE_NON_DEFINIE`, ce registre ou ce fils passe à la classe `CLASSE_DEFINIE`.
- `int getRegClassRec(Registre*)` donne la valeur de la classe du registre ainsi que celle de fils. Ainsi, cette fonction renvoie `CLASSE_NON_DEFINIE` si et seulement si le registre ou l'un de ses fils est de la classe `CLASSE_NON_DEFINIE`.
- `void setRegClassRec(Registre*, int classe)` affecte au registre et à l'ensemble de ses fils la classe `classe`.

3.2.3 Mémoire

Nos préoccupations en ce qui concerne la gestion de la mémoire virtuelle ont été les mêmes que celles au sujet des registres : l'accès à n'importe quel segment mémoire devait être simple et devait tenir compte des cases mémoire et les changer correctement s'il le fallait. Pour cela, il a fallu mettre en place plusieurs structures, à commencer par la mémoire de la machine. Elle est codée en `C`comme le montre le listing 5.

Listing 5 – Structure de la mémoire

```
typedef struct _byte{
    uint64_t virtualAddr;
    uint8_t val;
    uint8_t classe;
}Byte;

typedef  struct _memoire {
    uint64_t size;
    uint64_t sizeAllocatedMemory;
    Byte** tabBytes;
}Memoire;
```

Une mémoire est un tableau de `Bytes` ***triés par adresse mémoire*** , chacun connaissant cette adresse. Cette représentation permet une gestion relativement aisée de la mémoire, les accès se faisant par recherche dichotomique du `Byte` recherché, et l'écriture écrivant sur chacun des `Bytes` concernés, les créant si nécessaire.

La seconde structure est l'équivalent des registres pour la mémoire. Il s'agit des segments mémoire :

Listing 6 – Segment mémoire

```
typedef struct _segment{
    Memoire* mem;
    uint64_t virtualAddr;
    uint8_t taille; // en octet
}Segment;
```

Les fonctions importantes pour la gestion mémoire sont les suivantes :

- `int* getSegClassRec(Segment seg)` donne la classe d'un segment mémoire. Si le segment comporte des `Bytes` non définis dans le tableau, la fonction renvoie l'entier `CLASSE_NON_DEFINIE`.
- `uint64_t getSegVal(Segment seg)` donne la valeur du segment donné. Cette fonction ne peut être utilisée sur un segment de classe `CLASSE_NON_DEFINIE` car les cases mémoires ne sont peut-être même pas définies dans le tableau.
- `void setSegClassRec(Segment, int classe)` attribue à tous les `Bytes` du segment la classe `classe`. Si besoin la fonction crée les `Bytes` qui manquent.
- `uint64_t setSegVal(Segment seg, uint64_t val)` assigne à tous les `Bytes` d'un segment la valeur donnée et fait passer ces bytes à la classe `CLASSE_DEFINIE` si ils sont à la classe `CLASSE_NON_DEFINIE`. De même, si besoin il y a, la fonction crée les `Bytes` manquants.

3.2.4 Variable

Le type `Variable` est un *wrapper* permettant de manipuler indépendamment les `Registres` et les `Segments`. Il est défini en `C`de cette manière :

Listing 7 – Structure d'une Variable

```
typedef struct _variable{
    uint8_t type;
    Registre* reg;
    Segment seg;
}Variable;
```

Les fonctions utiles sont elles mêmes pour la plupart des *wrappers* des fonctions des `Registres` et des `Variables` :

- `uint64_t getVarVal(Variable)`

- `int getVarClassRec(Variable)`
- `uint64_t setVarVal(Variable, uint64_t)`
- `int getVarTaille(Variable)` donne la taille de la variable ***en bits***.
- `void setVarClassRec(Variable, int classe)`

3.2.5 Pile d'appel

Pour finir la simulation hardware, on a créé un modèle de pile d'appel. Elle utilise la `Memoire` pour stocker les informations sous forme d'une liste chaînée de `Segments`. Les fonctions essentielles sont :

- `void pushStack(Stack*, Variable, Registre* _RSP)`
- `void popStack(Stack*, Variable, Registre* _RSP)`

3.2.6 Processeur

Comme nous nous intéressons aux changement d'état de l'ordinateur nous avons du simuler le processeur. Nous avons choisi de représenter le processeur par 3 attributs, la structure de pile, un tableau de registres de flag contenant 9 flags, et un tableau de registres contenant 80 registres. Ce choix de représentation par tableau, permet de réutiliser le programme avec des processeurs avec différents flags, registres ; donc différents types de processeurs.

Listing 8 – structure du processeur

```
typedef struct _Processeur{ // 664 bytes
    Stack* stack;
    Registre*   tabRegistre[NOMBRE_REGISTRES];
    uint8_t     tabFlags[NOMBRE_FLAGS]
}Processeur;
```

Le diagramme de classe du hardware de la machine virtuelle est donné dans la figure 7.

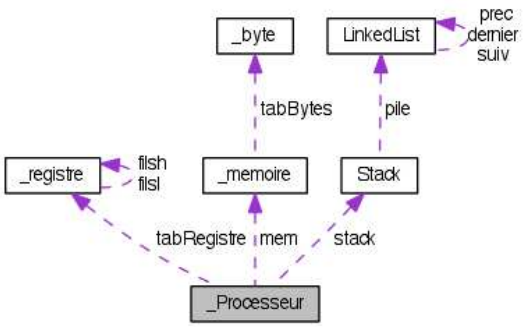


FIGURE 7 – Diagramme de classe du hardware de la machine virtuelle

3.3 Implémentation des instructions

3.3.1 Mode de représentation

Afin de pourvoir suivre l'état des registres, il faut connaitre l'effet des Instructions assembleur sur ces registres. Nous avons donc du créer une structure en mémoire qui permet de simuler l'action des instructions sur les différents registres. Nous avons essayé de simuler le paradigme de la POO dans la création de nouvelles fonctions d'instructions afin que l'on puisse en rajouter facilement. Pour prendre en compte les actions d'une nouvelle instruction, il suffit de créer les fonctions qui vont modifier les registres de flags de la même manière que l'instruction va le faire.

Listing 9 – structure d'instructions

```
typedef struct _instruction{
    int(*of_aux)(const Registre*,const Registre*,const Registre*);
    int(*cf_aux)(const Registre*,const Registre*,const Registre*);
    int(*af_aux)(const Registre*,const Registre*,const Registre*);
    int zf_aux;
    int pf_aux;
    int sf_aux;
    Registre* (*f)(Registre*,Registre*,Registre*,Processeur*,int);
}Instruction;
```

3.3.2 Ajout d'une instruction

La création des instruction doit s'appuyer sur la documentation Intel[**?**]. L'exemple suivant, va illustrer la procédure à suivre pour rajouter au désassembleur une Instruction.

Listing 10 – Création de l'instruction add

```
/* ---------------------- ADD ----------------------*/

static int of_add(const Registre* a, const Registre* b, const
   Registre* stub){
    uint64_t aa = getValeur(a);
    uint64_t bb = getValeur(b);
    uint64_t c = aa+bb;
    uint64_t p = pow(2, a->taille);
    if (p!= 0) {
        c = c % p;
    }
    if (c<aa) {
        return 1;
    } else {
        return 0;
    }
}
```

```
17 //Création ci-dessus de la fonction qui va modifier le flag d'
       overflow.
18
19 static int cf_add(const Registre* a, const Registre* b, const
       Registre* stub){
20     uint64_t aa = getValeur(a);
21     uint64_t bb = getValeur(b);
22     uint64_t c = aa+bb;
23     uint64_t p = pow(2, a->taille);
24     if (p!= 0) {
25         c = c \% p;
26     }
27     if (c<aa) {
28         return 1;
29     } else {
30         return 0;
31     }
32 }
33 //Création ci-dessus de la fonction qui va modifier le flag carry.
34
35 static int af_add(const Registre* a, const Registre* b, const
       Registre* stub){
36     uint64_t aa = getValeur(a) \% 8; // donne les 3 bits les plus
           faibles
37     uint64_t bb = getValeur(b) \% 8;
38
39     return (aa + bb)/ 8;
40 }
41
42 //Création ci-dessus de la fonction qui va modifier le flag adjust
       .
43
44 static Registre* f_add(Registre* destination, Registre* masque,
       Registre* stub , Processeur* proc, int lenInstr){
45     incr(_RIP, lenInstr);
46     uint64_t a = getValeur(destination);
47     uint64_t b = getValeur(masque);
48     uint64_t c = a+b;
49     setValeur(destination, c);
50     return destination;
51 }
52 //f est la fonction qui va effectuer virtuellement la même action
       que l'instruction sur le processeur.
53
54 Instruction* init_add(){
55     return newInstruction(of_add, cf_add, af_add, 1, 1, 1, f_add);
56 }
```

On crée donc les fonctions qui modifient chaque flag que l'on a décidé de considérer. Ensuite nous créons la nouvelle instruction. Dans la fonction `newInstruction`, les trois constantes prennent 1 si les flags (dans l'ordre) zéro, de parité et de signe peuvent être modifiés par l'instruction et 0 sinon. La fonction `f`, effectue elle la même action que l'instruction mais sur le processeur virtuel (quand cela est possible).
Cependant, pour que le suivi de l'exécution du programme soit efficace et utile, il faut que toutes les instructions rencontrées aient été créées auparavant. Tant que cela n'a pas été fait, dès que le désassembleur rencontrera une instruction inconnue, le calcul virtuel sera dénué de sens.

3.4 Fonctions utilisateur

Nous allons présenter ici les fonctions à utiliser lorsque l'on veut désassembler et rassembler des informations sur un exécutable.
Notons au passage que certaines fonctions produisent d'elles mêmes des fichiers `log` dont les adresses peuvent être changées dans le fichier `macro.h`.

3.4.1 L'objet `Desas`

Cette structure est un *wrapper* de la structure `DISASM` apportée par `Beaengine`. Elle a pour but de réunir les informations recueillies lors du chargement en mémoire d'un fichier exécutable. Elle contient :

- La structure `DISASM`
- Un pool (du type `Processeur*`)
- le début virtuel du bloc

Tout ses champs sont normalement initialisés lors de l'ouverture d'un exécutable.

3.4.2 Ouverture d'un fichier exécutable

La calibration de la structure `Desas` est effectuée grâce à la fonction

```
void load(Desasembleur*,Fichier*,int)
```

qui prend en paramètres

- le désassembleur à initialiser
- le fichier exécutable à ouvrir
- le type de fichier exécutable.

Ce dernier paramètre est un des entiers de l'énumération

```
enum TypeSystem{
        MACHO_64,
        ELF_32,
    };
```

Actuellement, la fonction d'ouverture ne gère que les fichiers Linux 32-bits et les fichiers Apple 64-bits.

3.4.3 Construction du graphe de flow

Une fois le désassembleur initialisé, l'étape suivante est de construire le graphe de flow du programme. Pour cela on invoque la fonction

```
Graphe * ControleFlow_entier(Desasembleur*)
```

qui prend en paramètre le désassembleur fraîchement initialisé. Si le travail du désassembleur s'arrête là, on peut appeler une fonction de simplification du graphe qui supprimera tous les nœuds ayant un unique père et au plus un fils et dont ni le père ni le fils n'ont respectivement qu'un seul père et qu'un seul fils. Cette fonction raccourcit les branches linéaires. On l'invoque par

```
Graphe * simplifieGraphe(Desasembleur*,Graphe*)
```

ou directement par

```
Graphe * ControleFlow_simplifie(Desasembleur*)
```

On peut représenter ce graphe en appelant la fonction

```
void enregistreGraphe(Graphe*,Fichier*)
```

qui prend en paramètres le graphe que l'on veut afficher et le fichier sur lequel on souhaite écrire.

3.4.4 Traitement du graphe

L'étape suivante consiste à utiliser le graphe de flow pour tirer des informations sur le programme. On peut par exemple invoquer la fonction

```
void optimizePool2(Graphe*,constProcesseur * initialPool)
```

prenant en paramètre un pool d'entrée et appliquant au graphe l'algorithme de propagation des constantes. Le résultat peut être affiché par la fonction

```
void enregistrePropagation(Fichier*,Graphe*)
```

qui enregistre dans un fichier la valeur du pool de chaque nœud. Dans ce fichier, nous avons numéroter les registres et nous n'utilisons pas la dénominations habituel. Cependant, la correspondance est faite dans le fichier `macro.h`[7].
On peut aussi directement demander au programme de supprimer les branches inutiles en appelant la fonction

```
void elagage(Graphe*,Processeur * poolInit)
```

7. Par exemple le registre eax correspond au registre n°1 et ebx au n°6.

Nous avons beaucoup commenté notre code et géneré une documentation Doxygen fourni où l'on peut trouver le descriptif détaillé de toutes les structures et fonctions. On le trouve à l'adresse suivante : http ://hurlebouc.github.com/desassembleur/.

3.5 Portabilité

Le fait d'utiliser différents OS a eu pour avantage de nous obliger à réaliser un désassembleur fonctionnant avec Mach-o et sur ELF. La principale difficulté a été lors de la réalisation de la fonction de chargement du point d'entrée en mémoire du programme. En effet, les structures des fichiers Mach-o et ELF n'étant pas les mêmes, il a fallu adapter cette fonction de chargement aux 2 types de fichier.

Annexes

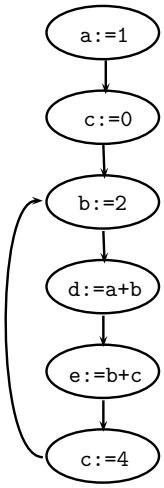
Méthodes d'analyse d'un programme désassemblé

Parmi les nombreuses méthodes qui permettent de faciliter la compréhension d'un code désassemblé, nous allons utiliser la propagation des constantes dans l'optique d'ensuite permettre d'utiliser l'extraction des sous-expressions. La méthode que nous allons expliciter est issue de l'article de Gary A. KILDALL [**?**].

Propagation des constantes

Ce que l'on appelle "propagation des constantes" est en fait quelque chose de facile à imaginer : c'est le recensement de toutes les variables utilisées par un programme dont on connait l'état à une étape de calcul donnée. Ce qui va nous intéresser dans la méthode c'est de pouvoir considérer les registres de flags, de travail, les cases mémoire comme des variables. On aura alors, pour chaque étape une idée de ce que va contenir les cases mémoire ciblées.

Nous allons utiliser tout le long de l'explication l'exemple de graphe de flow suivant (nous n'avons pas précisé ici les conditions de saut etc.).
L'algorithme que nous allons utiliser est celui décrit dans [**?**]. Nous le redonnons ici pour mémoire. Pour pouvoir utiliser en pratique l'algorithme décrit par [**?**], on va considérer le flow d'instructions (l'enchaînement), donc le programme comme un graphe. Les noeuds de ce graphe seront donc les instructions.
Posons quelques notations. $\mathfrak{N}$ est l'ensemble des nœuds du graphe. Soit V et C les ensembles des variables et des valeurs possibles. Soit $U \supset V \times C$ et $U \neq V \times C$. On appelle *pool* tout élément de $\mathfrak{P} = \mathscr{P}(U)$. En pratique un pool est donc l'ensemble des couples (variable,valeur prise) que l'on connait à une étape précise du déroulement du programme.
Notons $f : \mathfrak{N} \times \mathfrak{P} \to \mathfrak{P}$ la fonction *calcul* qui à tout nœud n et à tout pool p associe le pool des variables calculées au nœud n à partir du pool p. f correspond donc à l'approche que l'on va utiliser pour déterminer le pool lié à l'instruction qui va suivre à partir de l'instruction



courante.

On pose L le "front de propagation" du programme, c'est à dire la liste des couples $(n, p) \in \mathrm{N} \times \mathfrak{P}$ à évaluer. On pose p_i le pool associé au nœud i. Le pool p_i est initialisé à U.

1. Initialisation	L contient l'ensemble des nœuds associés à leur pool initial
2. Fin	Si L est vide, on s'arrête
3. Sélection d'un nœud	Sinon on choisit $(n, p) \in \mathrm{L}$ et on pose $\mathrm{L} \stackrel{\text{def}}{=} \mathrm{L} \setminus (n, p)$
4. Test d'inclusion	Si $p_n \subset p$ on retourne en 2.
5. Mise à jour du Pool	Sinon $p_n \stackrel{\text{def}}{=} p_n \cap p$
6. Appel récursif	On pose $\mathrm{L} \stackrel{\text{def}}{=} \mathrm{L} \cup \{(n', f(n, p_n) \mid n' \text{ est fils de } n\}$. On retourne en 2.

On peut prouver que cet algorithme termine et que le résultat ne dépend pas du choix effectué à l'étape 3.

Cela donne dans le cas de l'exemple le graphe de la figure 8
Cette méthode va nous permettre dans certains cas de lever des indéterminations dans le programme désassemblé. De plus, nous allons pouvoir l'appliquer aux registres de flags (dans la mesure de la connaissance en détails de l'implémentation des instructions processeur) et avoir une idée de plus en plus précise des effets du programme sans l'exécuter.

Extraction des sous-expressions

La méthode précédente peut être améliorée si on effectue une meilleure analyse de l'utilisation des variables. En effet, dans de nombreux programmes des calculs sont redondants de manière volontaire (obfuscation) ou involontaire. Cependant cela peut fausser les résultats de la propagation des constantes.

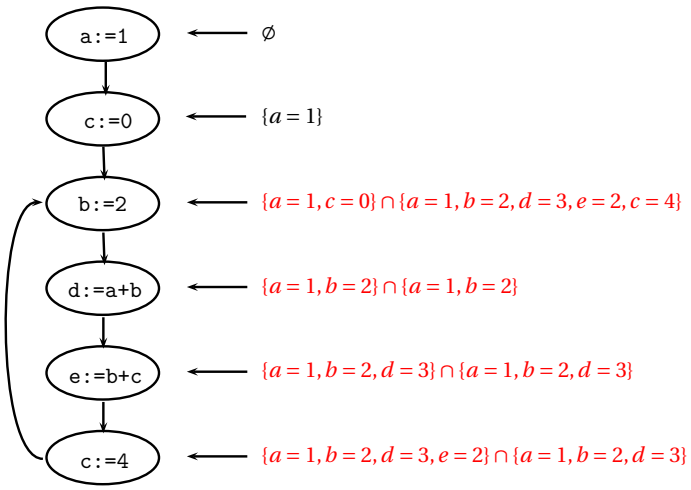


FIGURE 8 – En rouge les intersections des pools sur la boucle de retour lors du deuxième passage.

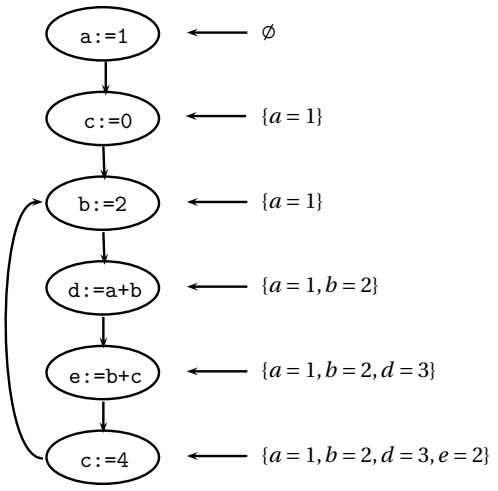


FIGURE 9 – Résultat final de la propagation des constantes sur l'exemple.

L'algorithme présenté dans la partie ci-dessous est toujours valide, à la condition d'étendre les opérateurs d'inclusion, d'intersection et d'union.
On peut rassembler les deux méthodes en un unique algorithme qui va donner pour chaque noeud du graphe l'ensemble des partitions des classes d'équivalence avec la valeur de cette classe si elle est connue. Cela va donner en pratique le graphe de la figure 10

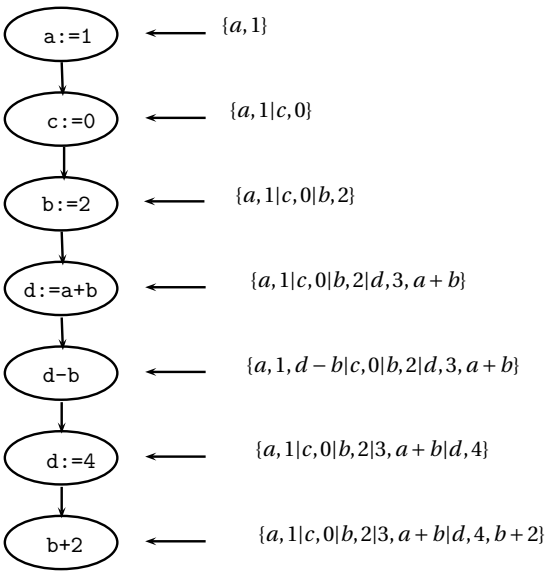


FIGURE 10 – Résultat de la propagation des constantes et de l'extraction des sous-expressions sur un exemple simple.

Correction de l'algorithme récursif de propagation des constantes

Nous allons démontrer ici que l'algorithme présenté par KILDALL est équivalent à notre implémentation. Rappelons que le point délicat concerne la consistance des appels récursifs dans notre algorithme : $f(n, p_n)$ n'est effectivement calculé que lors du choix de l'étape 3. Or p_n ayant pu être modifié par le traitement d'autres nœuds , $f(n, p_n)$ est comme "inconstant" à l'étape 6 lors de l'ajout à L des fils de n associés à $f(n, p_n)$.

Pour commencer, on peut réduire le problème au cas où on n'a qu'un seul nœud d'entrée. Cette réduction est assez intuitive et utilise la notion de nœud *muet*, c'est à dire un nœud n tel que $\forall p \in \mathfrak{P}, f(n, p) = p$. On pourra se reporter à l'article de KILDALL[**?**] pour une démonstration propre de la réduction.

On notera $\mathfrak{T} \subset \mathbb{N}$ l'ensemble des valeurs de temps, c'est à dire l'ensemble des étapes de l'algorithme et, $\forall (t, n) \in \mathfrak{T} \times \mathfrak{N}$, on notera p_n^t la valeur du pool du nœud n au temps t. Pour faciliter la démonstration, nous allons modifier l'algorithme pour que chaque élément de la liste L retienne le nœud père qui l'a appelé ainsi que le temps auquel il a été créé. On se convaincra

facilement que le déroulement de l'algorithme n'est en rien modifié.

On pose L le "front de propagation" du programme, c'est à dire la liste des couples $(n, \texttt{pool}, \texttt{pere}, t) \in \mathrm{N} \times \mathfrak{P} \times \mathrm{N} \times \mathfrak{T}$ à évaluer. Le pool p_n^0 est initialisé à U pour tout nœud n.

1. Initialisation	L contient le nœud d'entrée avec son pool initial. Le troisième membre est ici un nœud muet père du nœud d'entrée.
2. Fin	Si L est vide, on s'arrête. On incrémente t
3. Sélection d'un nœud	Sinon on choisi $(n, p, \_, \_) \in \mathrm{L}$ et on pose $\mathrm{L} \stackrel{\text{def}}{=} \mathrm{L} \setminus (n, p, \_, \_)$
4. Test d'inclusion	Si $p_n^{t-1} \subset p$, alors $\forall i \in \mathfrak{N}, p_i^t \stackrel{\text{def}}{=} p_i^{t-1}$ et on retourne en 2.
5. Mise à jour du Pool	Sinon $p_n^t \stackrel{\text{def}}{=} p_n^{t-1} \cap p$ et $\forall i \in \mathfrak{N} \setminus \{n\},\ p_i^t \stackrel{\text{def}}{=} p_i^{t-1}$
6. Appel récursif	On pose $\mathrm{L} \stackrel{\text{def}}{=} \mathrm{L} \cup \{(n', f(n, p_n^t), n, t \mid n' \text{ est fils de } n\}$. On retourne en 2.

Nous allons montrer le lemme suivant :

Lemme 1. *Dans l'algorithme de* KILDALL*, si, au moment t du choix de l'étape 3 du quadruplet $(n, \texttt{pool}, \texttt{pere}, t')$, le pool $f(\texttt{pere}, p^t_{\texttt{pere}})$ est **différent** du pool $\texttt{pool} = f(\texttt{pere}, p^{t'}_{\texttt{pere}})$, alors il existe un temps $t'' \in [\![t', t]\!]$ tel que la liste* L *ait comporté l'élément $(n, f(\texttt{pere}, p^t_{\texttt{pere}}), \texttt{pere}, t'')$.*

Démonstration. On sait que au moment t du choix du quadruplet $(n, \texttt{pool}, \texttt{pere}, t')$, on a $f(\texttt{pere}, p^t_{\texttt{pere}}) \neq \texttt{pool}$. Avant cet instant et après t', on a donc choisi au temps t'' un élément de L dont le premier élément était le nœud pere [8] et ce choix a conduit à modifier strictement le pool de pere en l'état $p^{t''}_{\texttt{pere}} = p^t_{\texttt{pere}}$. Cette modification a entrainé l'ajout à la liste L des nœuds de l'ensemble $\left\{(n', f(\texttt{pere}, p^t_{\texttt{pere}}), \texttt{pere}, t'' \mid n' \text{ est fils de } \texttt{pere}\right\}$. Or n fait partie des fils de pere ce qui achève la preuve. □

À présent démontrons le théorème suivant

Théorème 1. *Dans l'algorithme de* KILDALL*, au moment t de l'étape 3, on peut remplacer* pool *dans le quadruplet choisi $(n, \texttt{pool}, \texttt{pere}, t')$ par $f(\texttt{pere}, p^t_{\texttt{pere}})$.*

Démonstration. Plaçons nous au choix de l'étape 3 au temps t et choisissons l'élément $\tau_1 \stackrel{\text{def}}{=} (n, \texttt{pool}, \texttt{pere}, t')$. Si $\texttt{pool} = f(\texttt{pere}, p^t_{\texttt{pere}})$, alors on a rien à faire.
Sinon, le lemme 1 nous dit que il existe un temps t'' entre la création de ce quadruplet et son choix dans la liste tel que L comporte l'élément $\tau_2 \stackrel{\text{def}}{=} (n, f(\texttt{pere}, p^t_{\texttt{pere}}), \texttt{pere}, t'')$. Deux cas se présentent :

- Si au temps t, τ_2 n'est plus présent dans L, c'est à dire si il a été choisi au temps $t_3 < t$ alors, après son exécution, $p_n^{t_3} \subset f(\texttt{pere}, p^{t''}_{\texttt{pere}}) = f(\texttt{pere}, p^t_{\texttt{pere}})$.
  Or $\left(p_n^t\right)_t$ est décroissant. Donc, au temps t du choix de τ_1,
  $$p_n^{t-1} \subset p_n^{t_3} \subset f(\texttt{pere}, p^t_{\texttt{pere}}).$$

8. En effet, le seul moyen pour changer le pool d'un nœud est que ce nœud soit choisi dans L.

Or, toujours par décroissance des $\left(p_i^t\right)_t$, $p^t_{\texttt{pere}} \subset p_n^{t'}$ et par homomorphisme de f,

$$f(\texttt{pere}, p^t_{\texttt{pere}}) \subset f(\texttt{pere}, p^{t'}_{\texttt{pere}}) = \texttt{pool}.$$

Le test d'inclusion de l'étape 4 est donc valide et on retourne à l'étape 2 sans avoir modifié les valeurs des pools des nœuds à $t+1$. Mais on aurait obtenu exactement le même résultat si on avait remplacé `pool` par $f(\texttt{pere}, p^t_{\texttt{pere}})$.

- Si au temps t, τ_2 est encore présent, remplacer `pool` par $f(\texttt{pere}, p^t_{\texttt{pere}})$ dans τ_1 puis choisir τ_1(qui est maintenant égal à τ_2) dans la liste revient à choisir τ_2 avant τ_1 (car on a vu dans le point précédent que la liste L contenant deux fois τ_2 ou τ_1 revient au même) ce qui est valide. On se ramène donc au cas précédent.

□

Bibliothèque `Beaengine`

Toutes les informations ont été tirées du site du créateur de la librairie[**?**]. Nous ne présenterons ici que les outils que nous avons utilisés.

Objet `DISASM`

Le logiciel utilise une structure `C` qui réunit toutes les informations nécessaires au désassemblage (données *in*). Elle permet également au programme de renvoyer des informations (données *out*).

Utilisation Cet objet est relativement simple d'utilisation. Une fois le programme que l'on veut désassembler chargé en mémoire, on renseigne l'objet `DISASM` avec l'adresse mémoire du premier octet du programme chargé, puis on applique à `DISASM` la fonction `int:Disasm(*DISASM)` qui désassemble la première instruction du programme. Il faut donc appliquer la fonction autant de fois qu'il y a d'instructions.

Structure de `DISASM` Ici on va voir comment utiliser un objet `DISASM`. Le listing 11 donne sa définition en `C`.

Listing 11 – Structure `DISASM`

```
typedef struct _Disasm {
   UIntPtr EIP;
   UInt64 VirtualAddr;
   UIntPtr SecurityBlock;
   char CompleteInstr[INSTRUCT_LENGTH];
   UInt32 Archi;
   UInt64 Options;
   INSTRTYPE Instruction;
   ARGTYPE Argument1;
   ARGTYPE Argument2;
   ARGTYPE Argument3;
   PREFIXINFO Prefix;
```

```
   UInt32 Reserved_[40];
}DISASM;
```

On va maintenant décrire tous les champs et leur utilité. Dans un premier temps regardons les champs d'entrée.

[in] EIP est l'adresse en mémoire vive du programme chargé.

[in] VirtualAddr est l'adresse où l'on souhaiterait voir l'instruction. Elle permet de rendre le désassemblage indépendant de son chargement en mémoire. Il ne s'agit ni plus ni moins que d'une translation du repère : si e est le point d'entrée réel et e' l'entrée virtuelle, on a $e - e'$ reste constant pendant tout le long du désassemblage du l'ensemble du programme. De plus, lorsque `Beaengine` donnera les adresses des sauts ou des `call` par exemple, il les affichera dans le repère virtuel « translaté ».
Ce champ est facultatif. S'il n'est pas spécifié, il est égal à `EIP`.

[in] SecurityBlock donne le nombre maximal d'octets que le `Beaengine` est autorisé à lire. Nous nous en sommes servi pour limiter les plages d'instructions que `Beaengine` pouvait désassembler. Par exemple, pour empêcher `Beaengine` de dépasser l'espace mémoire dans lequel est stocké le programme à désassembler, `SecurityBlock` contient le nombre d'octets du programme.

[in] Archi spécifie l'architecture du processeur (32-bits par defaut).

[in] Options permet de paramétrer la sortie du désassemblage comme le langage assembleur souhaité (nasm, gas, etc.). Les constantes sont les suivantes :

- `NoTabulation = 0x0`
- `Tabulation = 0x1`
- `MasmSyntax = 0x000`
- `GoAsmSyntax = 0x100`
- `NasmSyntax = 0x200`
- `ATSyntax = 0x400`
- `PrefixedNumeral = 0x10000`
- `SuffixedNumeral = 0x00000`
- `ShowSegmentRegs = 0x01000000`

La structure fournit également des champs permettant de lire les résultats du désassemblage.

[out] CompleteInstr renvoie l'instruction complète en assembleur. Elle dépend des options passées à `DISASM` .

[out] Instruction donne des informations sur l'instruction lue. Elle est du type INSTRTYPE (voir le listing 12 et le détail des champs).

[out] Argument1, Argument2, Argument3 donnent des informations sur les registres utilisés dans les instructions. Ils sont du type ARGTYPE (listing 13) que nous détaillerons plus loin.

Structures annexes On va maintenant voir le détail des champs de la structure qui contient les informations que l'on récupère d'une instruction.

Listing 12 – Structure INSTRTYPE

```
typedef struct INSTRTYPE {
    Int32 Category;
    Int32 Opcode;
    char Mnemonic[16];
    Int32 BranchType;
    EFLStruct Flags;
    UInt64 AddrValue;
    Int64 Immediat;
    UInt32 ImplicitModifiedRegs;
 }INSTRTYPE;
```

[out] Category indique si l'instruction est une instruction logique, arithmétique, de transfert de données etc...

[out] Opcode contient l'opcode sur 1,2 ou 3 octets.

[out] Mnemonic renvoie le mnémonique de l'instruction en format ASCII suivi d'un espace.

[out] BranchType indique, si l'instruction décodée est une instruction de saut, quel est le type de saut rencontré.

[out] Flags renvoie une structure qui rassemble les informations sur les registres EFLAGS. Pour chacun, cela permet de savoir si le registre a été modifié, testé, remis à zéro, etc.

[out] AddrValue indique dans le cas où l'instruction décodée est un saut l'adresse du saut quand elle est facilement déterminable. Sinon, la valeur par défaut est 0.

[out] Immediat contient la constante utilisée par l'instruction si elle en a utilisé une.

[out] ImplicitModifiedRegs donne des informations lorsque des instructions (comme push 0 qui va modifier ESP) modifient de façon implicite un registre.

Listing 13 – Structure ARGTYPE

```
typedef struct ARGTYPE {
    char ArgMnemonic[32];
    Int32 ArgType;
    Int32 ArgSize;
    Int32 ArgPosition;
    UInt32 AccessMode;
    MEMORYTYPE Memory;
    UInt32 SegmentReg;
}ARGTYPE;
```

[out] ArgMnemonic renvoie un mnemonique en format ASCII lorsque cela est possible (pour désigner un registre usuel par exemple).

[out] ArgType renseigne sur le type de l'argument. C'est-à-dire, indique si il s'agit d'un registre, d'un emplacement mémoire ou d'une constante.

[out] ArgSize renvoie la taille de l'argument.

[out] ArgPosition renvoie la position du mode utilisé du registre utilisé en 8 bits : 1 si le registre est utilisé en position haute (ie : ah,ch etc...) et 0 sinon.

[out] AccessMode indique si l'argument a été modifié ou pas.

[out] Memory renvoie une structure dans le cas de l'utilisation d'un espace mémoire. Cette structure contient les informations permettant d'utiliser la formule : `BaseRegister + IndexRegister*Scale + Displacement` (ie : BaseRegister, IndexRegister, Scale et displacement).

[out] SegmentReg contient, dans le cas de l'accés mémoire, le registre de segment utilisé.

Exemples d'utilisation

Le listing 14 est un exemple dans lequel le programme va se désassembler lui-même sur 20 instructions. On a ici un désassemblage linéaire (linear sweep).

Listing 14 – Premier exemple

```
#include <stdio.h>
#include <string.h>
#include "BeaEngine.h"
int main(int argc, char* argv []){
    /* ======== cree un DISASM */
```

```
    DISASM MyDisasm;
    /* ======== met tous les champs a 0*/
    memset (&MyDisasm, 0, sizeof(DISASM));
    /* ======== donne le point d entree */
    MyDisasm.EIP = &main;
    /* ======== donne les options d affichage*/
    MyDisasm.Options = Tabulation + NasmSyntax + PrefixedNumeral +
        ShowSegmentRegs;
    /* ======== specifie l architecture*/
    MyDisasm.Archi = ARCHI_PROC;
    /* ======== desassemble sur 20 instructions */
    int len, i, Error = 0;
    while ((!Error) && (i<20)){
        len = Disasm(&MyDisasm); // recupere la taille de l
            instruction
        if (len != UNKNOWN_OPCODE) {
            printf("%\ns",MyDisasm.CompleteInstr);
            MyDisasm.EIP = MyDisasm.EIP + len;
            i++;
        }
        else {
            Error = 1;
        }
    }
    return 0;
}
```

On peut aussi, au lieu de se limiter à 20 instructions, ne vouloir désassembler qu'une certaine quantité d'octets. On peut pour cela utiliser le champ `SecurityBlock` qui contiendra la taille de la suite d'octets que l'on souhaite décompiler. Dans ce cas la boucle de désassemblage sera

```
while(!Error){
    MyDisasm.SecurityBlock = finProg - MyDisasm.EIP;
    len = Disasm(&MyDisasm);
    if (len != UNKNOWN_OPCODE) {
        printf("%\ns",MyDisasm.CompleteInstr);
        MyDisasm.EIP = MyDisasm.EIP + len;
        if(MyDisasm.EIP>=finProg){
            printf("fin du programme");
            Error = 1;
        }
    }
    else if(len = OUT_OF_BLOCK){
       printf("beanengine n est pas autorisé à aller plus loin\n")
          ;
       Error = 1;
    } else {
       printf("opcode inconnu");
    }
```

```
}
```

Il est important de noter que `len` n'est égal à `OUT_OF_BLOCK` que lorsque `MyDisasm.SecurityBlock` passe de positif strictement à négatif strictement. C'est pour cela qu'on utilise la condition `MyDisasm.EIP>=finProg`.

Si l'on souhaite que les adresses mémoire affichées par `Beaengine` ne dépendent pas de l'endroit où a été chargé le programme, on peut utiliser les adresses virtuelles `VirtualAddr`. Un exemple d'un tel programme est donné dans le listing suivant.

Listing 15 – Utilisation des adresses virtuelles

```
#include <stdio.h>
#include <string.h>
#include "BeaEngine.h"
int main(int argc, char* argv []){
    DISASM MyDisasm;
    memset (&MyDisasm, 0, sizeof(DISASM));

    /* ======== donne le point d entree reel et virtuel*/
    MyDisasm.EIP = &main;
    MyDisasm.VirtualAddr = 0x1000000;
    MyDisasm.Options = Tabulation + NasmSyntax + PrefixedNumeral +
        ShowSegmentRegs;
    MyDisasm.Archi = ARCHI_PROC;
    int len, i, Error = 0;
    while ((!Error) && (i<20)){
        len = Disasm(&MyDisasm); // recupere la taille de l
            instruction
        if (len != UNKNOWN_OPCODE) {
            printf("%\ns",MyDisasm.CompleteInstr);
            MyDisasm.EIP += len;
            MyDisasm.VirtualAddr += len;
            i++;
        }
        else {
            Error = 1;
        }
    }
    return 0;
}
```

On peut également utiliser le champ `Instruction` pour étudier plus en détails les instructions à chaque ligne. Par exemple, voila une fonction désassemblant en suivant les sauts inconditionnels. On utilise les champs `AddrValue` et `BranchType` de `Instruction`. On notera que `AddrValue` est dans le repère translaté (celui des adresses virtuelles).

Listing 16 – Suivi des sauts inconditionnels

```
void desassemblage_saut(DISASM* prog) {
```

```
   int erreur = 0;
   unsigned long finProg = prog->EIP + prog->SecurityBlock;
   int len;

   while (!erreur) {
      prog->SecurityBlock = finProg - prog->EIP;
      len = Disasm(prog);
      if (len == UNKNOWN_OPCODE) {
         erreur = 1;
         printf("Code inconnu\n");
      } else if (len == OUT_OF_BLOCK) {
         erreur = 1;
         printf("Fin du bloc\n");
      } else {
         unsigned long adresseIni = prog->VirtualAddr;
         printf("(0x%lx) \t 0x%lx \t %s \t (0x%lx)\n", prog->EIP,
            adresseIni, prog->CompleteInstr, prog->Instruction.
            AddrValue);
         unsigned long IP = adresseIni + len;
         switch (prog->Instruction.BranchType) {

            case JmpType:// un type de branche
               prog->VirtualAddr = prog->Instruction.AddrValue;
               prog->EIP += prog->VirtualAddr - (long) adresseIni;
               break;

            default:
               prog->VirtualAddr += len;
               prog->EIP += len;
         }
         if (prog->EIP >= finProg) {
            printf("fin de la lecture");
            erreur = 1;
         }
      }
   }
}
```

Remarques pratiques

L'utilisation de `Beaengine` nous a permis de mettre en évidence certains comportements étranges de la bibliothèque. Ces apparentes anomalies apparaissent majoritairement lorsque l'on veut connaître des informations sur les arguments des instructions ne prenant en paramètre qu'un seul argument. Dans ce cas, la valeur du premier (et unique) argument se trouve dans le deuxième argument de `Beaengine` (`disas.Argument2`).

Programme exemple

Nos exemples s'articulent autour du désassemblage du programme 17 qui a été obfusqué en modifiant son graphe de flow (programme 18) et en indéterminant ses branches (programme 19)

Listing 17 – Programme non obfusqué

```
main:
        mov eax, 3
        mov ebx, 36
        add eax, ebx
        imul ecx, ebx, 2
        sub eax, ebx
        hlt
```

Listing 18 – Obfuscation par modification du graphe de flow

```
main:
        mov eax, 3
        jmp a
b:
        add eax, ebx
        jmp c
e:
        mov eax, ecx
        and esp, ebp
d:
        sub eax, ebx
        hlt
a:
        mov ebx, 36
        jmp b
c:
        imul ecx, ebx, 2
        jmp d
```

Listing 19 – Obfuscation par indétermination des branches

```
main:
        mov eax, 3
        cmp eax, 3
        je a
b:
        add eax, ebx
        cmp eax, 39
        je c
e:
        mov eax, ecx
        and esp, ebp
```

```
d:
        sub eax, ebx
        cmp eax, 3
        jne e
        hlt
a:
        mov ebx, 36
        cmp ebx, 36
        je b
c:
        imul ecx, ebx, 2
        cmp ecx, 72
        je d
```

Références

[1] Beaengine, disassembler library x86 x86-64 (ia32 and intel64).

[2] *Intel 64 and IA-32 Architecture Software Developer's Manual*, October 2011.

[3] Saumya Debray Cullen Linn. Obfuscation of executable code to improve resistance to static disassembly. *Department of Computer Science University of Arizona*, 1999.

[4] Ludovic Mé Jean-Marie Borello. Code obfuscation techniques for metamorphic viruses. *J Comput Virol*, 2008.

[5] Gary A. Kildall. A united approach to global program optimization.