ÉCOLE POLYTECHNIQUE
DE MONTRÉAL

INF6302 - RÉ-INGÉNIERIE DU LOGICIEL

Conception d'un analyseur de binaire ARM

Rapport final

Auteur :
Adrien VERGÉ

15 avril 2013

Superviseurs :
Ettore MERLO
Thierry LAVOIE

Table des matières

1 Présentation du projet

1.1 Résumé

Nous proposons de réaliser un programme permettant l'analyse de binaires exécutables compilés pour l'architecture ARM. Plus précisément, cet analyseur permet d'extraire les informations nécessaires à la reconstruction de l'architecture du programme. Cette reconstruction architecturale comprend la détection des instructions de branchement, la récupération des fonctions utilisées par le développeur, ainsi que les liens entre ces différentes fonctions. Sous certaines conditions (énoncées dans la section 1.4), la reconstruction est fidèle à la structure du programme étudié, et notre analyseur est capable de tracer le graphe d'appel ainsi que le graphe de flot de contrôle (CFG).

Bien que le but de ce projet ne soit pas de *décompiler* des exécutables binaires, il pose les bases pour une étude de ce genre. En effet, il permet de délimiter les fonctions et les branchements inter-procéduraux, ce qui serait une première étape dans la récupération complète du code source d'un programme. Pour le lecteur intéressé, des utilitaires de décompilation existent déjà à ce jour et sont mentionnés dans la section 2.2.

1.2 Enjeux

L'analyse de binaires compilés que nous proposons dans ce projet est un exemple de rétro-ingénierie informatique, appliquée à l'architecture ARM. Elle permet de comprendre le fonctionnement d'un programme « en boîte noire », pour éventuellement en créer une version adaptée à ses besoins, lorsque cela est autorisé d'un point de vue juridique (voir la section 2.1).

La multiplication des machines utilisant l'architecture ARM, en partie due au succès des appareils mobiles, rend légitime l'existence d'outils permettant cette étude. Le portage de pilotes de périphériques d'appareils embarqués est un exemple d'application d'un tel outil, dans le cas où le fabricant n'avait pas prévu l'utilisation de son appareil avec d'autres systèmes (comme GNU/Linux), mais est d'accord avec le portage par un développeur tiers. Avec la prolifération des tablettes numériques et téléphones intelligents fonctionnant sous Windows, et n'ayant pas de pilotes à code ouvert, la question de la rétro-ingénierie autorisée mérite d'être abordée.

1.3 Objectifs

Dans le cadre de ce projet d'une session, nous nous fixons les objectifs suivants :

- récupérer les fonctions utilisées dans le programme source : adresses de début et de fin, nom quand il est disponible ;
- détecter les branchements au sein des fonctions : appels inter-procéduraux ou sauts, conditionnels ou pas, adresses statiques ou dynamiques ;

– détecter les appels système.

En utilisant ces informations, notre programme doit être en mesure de :

– tracer le graphe d'appel ;
– tracer le graphe du flot de contrôle (CFG) de chaque fonction.

1.4 Hypothèses

Conteneur Les programmes compilés que nous étudierons seront au format ELF (*Executable and Linkable Format* [16]). Nous avons fait ce choix car ce format de conteneur a une spécification ouverte et est très bien documenté. De plus, c'est le format des exécutables sous GNU/Linux, ce qui est un avantage car de nombreux programmes de ce type sont disponibles.

Architecture Le contenu d'un exécutable dépend de l'architecture pour laquelle il est compilé. En effet, le code machine n'est pas le même selon sur quel processeur le programme est exécuté. Pour cette raison, nous devons faire un choix. C'est l'architecture ARM 32-bit version 5 qui a été retenue, pour les raisons suivantes.

– Contrairement au x86, les instructions sont à longueur fixe. Des instructions à longueur variable sont beaucoup plus dures à extraire car elles ne sont pas alignées en mémoire. Le choix de la version 5 est dû au fait qu'elle précède l'introduction des instructions *Thumb* dans le jeu ARM. Les instructions *Thumb* étant codées sur 16 bits, le mélange de *Thumb* et d'instructions régulières complique l'analyse du code binaire de la même façon que des instructions à longueur variable.
– C'est une architecture RISC (*Reduced Instruction Set Computer*), assez facile à étudier car elle comprend un nombre limité d'instructions. Dans le jeu d'instructions ARMv5, nous en avons compté 147 [6].

Librairies dynamiques La majorité des programmes utilisent des librairies partagées, pour éviter de contenir des portions de code déjà définies dans d'autres fichiers. Au lancement de l'application, ces librairies sont chargées en mémoire pour pouvoir être utilisées. Ce type de chargement dynamique sera exclu car il complexifierait le code du projet, sans ajouter aucun concept intéressant. En effet, n'importe quel programme à chargement dynamique peut avoir son équivalent statique : ce ne sont que des sections brutes à inclure dans l'exécutable et à ajouter dans la table des symboles. L'hypothèse retenue ici sera donc d'utiliser des programmes compilés statiquement.

Langage source On utilisera des programmes compilés écrits en C, car ce langage est un des plus utilisés : par exemple, dans la plupart des systèmes d'exploitation. La traduction en code machine de ce langage bas niveau n'est pas trop éloignée du code originel. Ceci permettra un déverminage plus aisé, notamment lorsqu'il faudra comparer le code source et la reconstruction générée par notre analyseur.

Compilateur et optimisation Le compilateur utilisé pour générer l'exécutable influe grandement sur l'organisation du code binaire. Pour cette raison, nous nous limiterons à des programmes compilés avec le compilateur C gcc [12], qui est le compilateur libre le plus utilisé.

De même, les optimisations du compilateur rendent les binaires plus complexes à analyser (voir section 3.4). Dans un premier temps, nous nous sommes limités à des exécutables compilés avec l'option `-O0` de gcc [12], ce qui désactive les optimisations. Depuis, les réarrangements du binaire dûs à l'optimisation ont été pris en compte par notre analyseur. Celui-ci retrouve les fonctions des exécutables compilés avec `-O2` (qui est l'option par défaut). Nous n'avons pas essayé avec `-O3`.

1.5 Lien avec ma recherche

Dans le cadre de ma maîtrise recherche à l'École Polytechnique de Montréal, sous la direction de Michel Dagenais, je travaille sur des processeurs ARM. Mon but est de configurer des modules matériels intégrés à certains nouveaux processeurs, afin de générer des traces de programmes pour la suite de traçage LTTng [3]. Ce projet de ré-ingénierie du logiciel est donc l'occasion pour moi de me familiariser avec les processeurs ARM, tout en expérimentant dans le domaine de la ré-ingénierie et rétro-ingénierie.

2 Cadre théorique

2.1 Aspect légal

La légalité des méthodes mises en œuvres lors de ce projet est absolument requise. La rétro-ingénierie, et plus particulièrement la décompilation de programmes est sujette à controverse. Pour être sûr de travailler dans un cadre légal, nous avons fait les recherches nécessaires en matière de droit canadien.

Il y a plusieurs usages légaux de la rétro-ingénierie : en cas de code source perdu, pour la recherche, ou plus souvent pour l'interopérabilité. Par exemple, il est autorisé de décompiler un pilote propriétaire pour rendre son matériel compatible avec son système d'exploitation.

Voilà un extrait de la loi sur le droit d'auteur de la législation canadienne, article 30.6 [7].

> « *Ne constitue pas une violation du droit d'auteur le fait, pour le propriétaire d'un exemplaire — autorisé par le titulaire du droit d'auteur — d'un programme d'ordinateur, ou pour le titulaire d'une licence permettant l'utilisation d'un exemplaire d'un tel programme :*
>
> *a) de reproduire l'exemplaire par adaptation, modification ou conversion, ou par traduction en un autre langage informatique, s'il établit que la copie est destinée à assurer la compatibilité du programme avec un ordinateur donné, qu'elle ne sert qu'à son propre usage et qu'elle a été détruite dès qu'il a cessé d'être propriétaire de l'exemplaire ou titulaire de la licence, selon le cas ;*
>
> *b) de reproduire à des fins de sauvegarde l'exemplaire ou la copie visée à l'alinéa a) s'il établit que la reproduction a été détruite dès qu'il a cessé d'être propriétaire de l'exemplaire ou titulaire de la licence, selon le cas.* »

Dans le cadre de ce projet, nous travaillerons exclusivement sur des logiciels dont la licence autorise l'étude du code binaire : des programmes de test de notre composition, et les utilitaires du projet GNU Coreutils [10] (voir la section 3.1.3).

2.2 État de l'art

Analyse de binaires Des outils existent pour étudier les programmes compilés. Leur utilisation simplifie énormément la tâche car ils essaient de retrouver les fonctions et structures, et calculent automatiquement les adresses de saut des branchements. Sous GNU/Linux, on peut citer gdb [11] pour l'analyse dynamique et objdump [9] pour l'analyse statique. Pour les programmes Windows, les outils IDA [14], WinDbg [15] et OllyDbg [17] sont les références pour l'analyse de binaires.

Décompilation La décompilation est le fait de recréer le code source d'un logiciel à partir de l'exécutable binaire. Pour du code partiellement compilé, tel que le

bytecode Java ou Python, il existe des méthodes de décompilation qui reposent des bases théoriques. Lorsque le code est entièrement compilé, comme c'est le cas pour le C, le problème n'a pas de solution théorique. En effet, la compilation entraîne une perte d'information irréversible comme par exemple les noms des variables et les structures de données. En pratique, on arrive à retrouver certaines informations à partir d'un binaire, et dans certains cas à obtenir un code source qui, recompilé, a le même comportement que le binaire initial.

Parmi les logiciels tentant de décompiler des programmes, on peut citer Boomerang [1], projet de décompilateur à code ouvert, ou dcc [8] qui fait l'objet d'une thèse.

3 Réalisation du projet

3.1 Préparations préliminaires

3.1.1 Structure générale

Nous avons choisi d'écrire l'analyseur de binaire ARM en C, essentiellement pour des raisons de performance.

Il doit être capable d'ouvrir un fichier binaire, l'étudier, et reconstruire l'image du programme et des fonctions dans un format approprié. Pour cela, nous décomposerons l'analyseur en parties ayant des tâches indépendantes. Elles sont représentées dans la figure 1.

Machine virtuelle Cette partie ouvre le programme à analyser, reconnaît s'il s'agit d'un exécutable au bon format (ELF, ARM 32-bit), et charge ensuite les sections exécutables en mémoire. Elle exporte une fonction permettant de lire une instruction à une adresse donnée. Point important, elle est aussi capable d'extraire à quelle adresse le programme commence. Enfin, cette partie lit la table des symboles. Lorsqu'elle est présente, cette table contient entre autres les noms associés à certaines adresses. Le cas échéant, ces symboles seront utilisés pour nommer les fonctions détectées.

Historique des adresses Cette partie définit une classe qui garde en mémoire l'ensemble des adresses déjà visitées, afin de ne pas relire plusieurs fois les mêmes instructions. Elle est hautement optimisée pour éviter les longues boucles de comparaison. Elle dispose de mécanismes de création et de fusion d'intervalles, qui rendent les tests (appartenance ou non à l'historique) très efficaces.

Désassembleur Cette partie contient un ensemble de fonctions spécifiques à l'architecture ARMv5 32-bit. Elle est utilisée par la partie décompilation pour caractériser les instructions de branchement, et pour calculer les adresses de saut.

Décompilateur Cette partie lit les instructions à partir de la machine virtuelle, et décode celles qui sont intéressantes. Les instructions les plus intéressantes sont les branchements (sauts, appels, retours), mais les autres types sont aussi analysés car ils peuvent apporter de l'information. Par exemple, les instructions NOP peuvent renseigner sur la fin des fonctions. De même, une lecture ou écriture à une adresse particulière indique généralement que le code à cette adresse n'est pas fait pour être exécuté.

Une fois toutes les instructions d'intérêt enregistrées, on associe chaque branchement à un saut, un retour ou un appel de fonction. Cette décision est faite par un algorithme bien spécifique et développé de manière empirique ; il est décrit en détail dans la section 3.2.3. C'est aussi à ce moment que l'on détermine les adresses de début et de fin des fonctions.

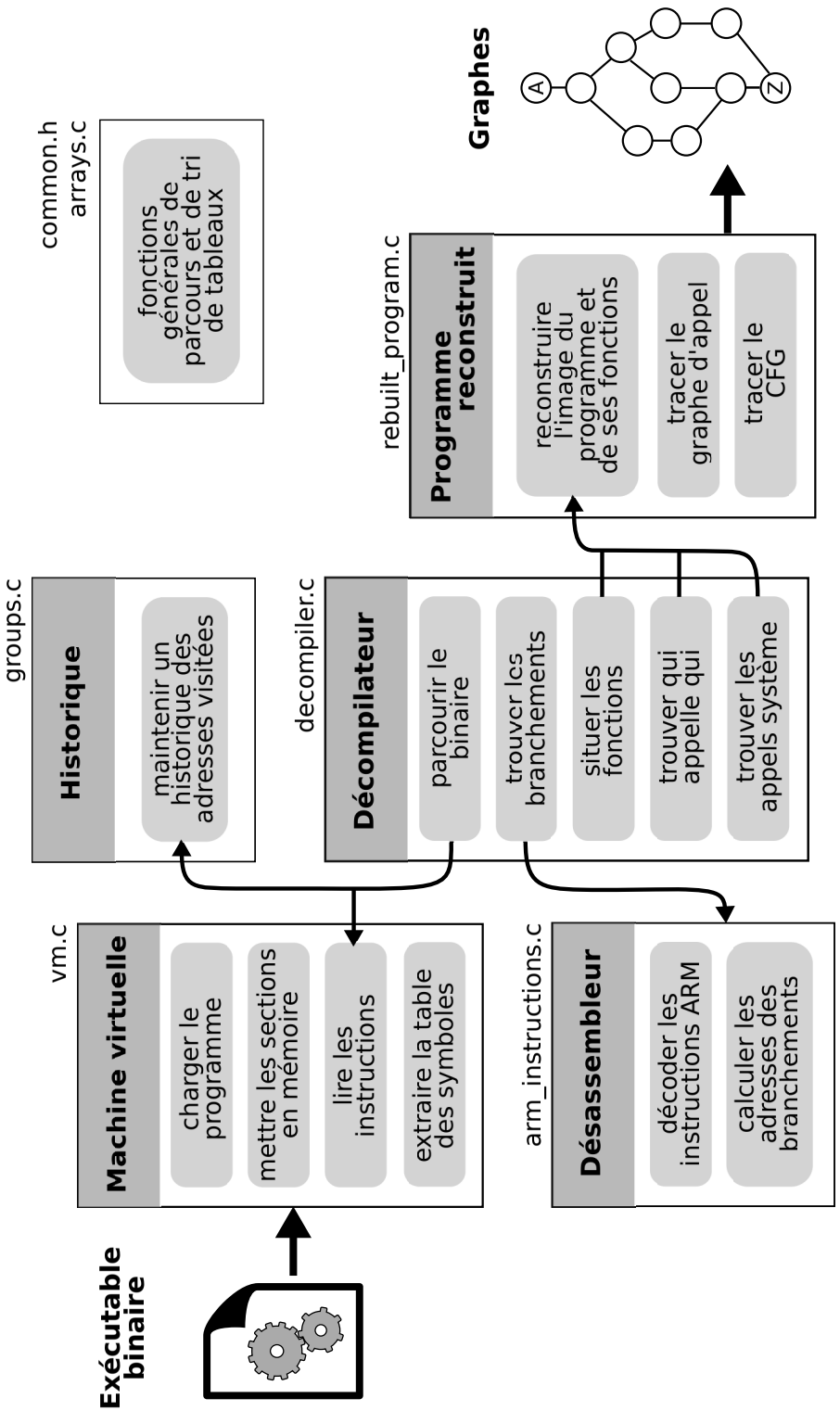


FIGURE 1 – Structure générale de l'analyseur

Enfin, cette partie parcourt le code binaire de chaque fonction reconstruite à la recherche des appels système.

Programme reconstruit Cette partie définit des structures de données pour représenter l'image du programme, ses branchements et ses fonctions. Elle inclut des méthodes pour modifier facilement ces structures, et les analyser pour générer le graphe d'appel et le CFG. Le format d'affichage peut être du simple texte, ou bien un fichier structuré adapté à l'affichage d'un graphe avec GraphViz [2].

Fonctions générales D'autres fichiers définissent des macros pour gérer efficacement les tableaux de taille dynamique, et des fonctions pour effectuer des algorithmes de tri (tri fusion).

3.1.2 Outils utilisés

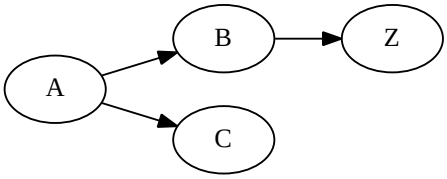
Compilateur croisé Pour tester notre analyseur, il est nécessaire de pouvoir créer des binaires ARM arbitraires. Depuis un ordinateur standard avec une architecture x86, il faut utiliser un compilateur croisé, capable de générer du code machine pour l'architecture ARM. Nous avons opté pour le compilateur gcc de la suite GNU EABI [12]. Sous Debian, la commande correspondante est `arm-linux-gnueabi-gcc`.

Désassembleur Un désassembleur comme objdump [9] permet de traduire les instructions binaires d'un programme en assembleur lisible. Un tel outil nous permet d'une part de vérifier que notre analyseur décode les instructions correctement, mais aussi de vérifier que la structure du programme que nous extrayons est correcte.

Librairie libelf La *Free Software Foundation* fournit une librairie qui simplifie l'étude de binaires au format ELF : la libelf [13]. Cette librairie permet de récupérer les adresses des sections exécutables d'un binaire ELF, ainsi que les symboles du programme, tout en s'affranchissant des difficultés liées à la plateforme : 32 ou 64 bits, petit ou grand boutisme, etc.

Générateur de graphes GraphViz [2] est un logiciel de génération de graphes à partir de fichiers en langage DOT. Voici un exemple de graphe généré par GraphViz :

```
digraph callgraph {
    A -> B -> Z;
    A -> C;
}
```



Cet outil sera utile pour générer les graphes de flot de contrôle et graphes d'appel, à partir d'une sortie au format DOT de notre analyseur.

3.1.3 Création de binaires de test

Binaires simplistes Les premiers binaires analysés sont des programmes extrêmement simples : il s'agit de `helloworld` et `helloworld-stdlib`. Le premier ne contient que six fonctions élémentaires, le second est comme le premier, mais compilé avec la librairie C standard. Leur compilation s'effectue au moyen des commandes suivantes :

```
arm-linux-gnueabi-gcc -Wall -O0 -static -march=armv5 \
    -nostdlib -o helloworld helloworld.c
arm-linux-gnueabi-gcc -Wall -O0 -static -march=armv5 \
    -o helloworld-stdlib helloworld-stdlib.c
```

Binaires réalistes Afin de tester de véritables programmes, nous avons compilé la suite GNU Coreutils [10], qui contient les classiques ls, echo, cat, wc, cp... Pour générer les binaires au bon format, il est nécessaire de configurer le projet avec les bonnes options de compilation :

```
./configure --host=arm-linux-gnueabi \
    --target=arm-linux-gnueabi CFLAGS=-O0 LDFLAGS=-static
make -j 8
```

Autres binaires Les binaires ainsi créés sont inclus dans l'archive du livrable, pour que le lecteur puisse les utiliser directement. Néanmoins, il est possible d'essayer n'importe quel autre programme, il suffit pour cela de le compiler avec les bonnes options : utiliser un compilateur croisé et indiquer à l'éditeur de lien de générer des librairies statiques.

3.2 Implémentation

3.2.1 Machine virtuelle

Le code de cette partie est présent dans le fichier `vm.c`.

Format ELF Le format *Executable and Linkable Format* [16], présenté dans la figure 2, est le standard sur les systèmes Unix. Un binaire ELF commence par une en-tête, puis le reste du fichier peut être vu comme une suite de *segments*, ou une suite de *sections* selon le point de vue. Nous adopterons la vision des sections, qui est plus adaptée à notre problème.

Adresse d'entrée L'adresse de la première instruction du programme, appelée point d'entrée, doit être récupérée à partir de l'en-tête ELF. Généralement, c'est `0x8150` pour un programme compilé avec la librairie C.

Chargement en mémoire Le but de cette partie de la machine virtuelle est de charger en mémoire les sections qui contiennent du code machine. Pour cela, notre programme parcourt l'ensemble des sections à la recherche de celles de type `SHT_PROGBITS` et qui possèdent les drapeaux `SHF_ALLOC`, correspondant respectivement à du code exécutable et à une partie devant être chargée en mémoire pour l'exécution.

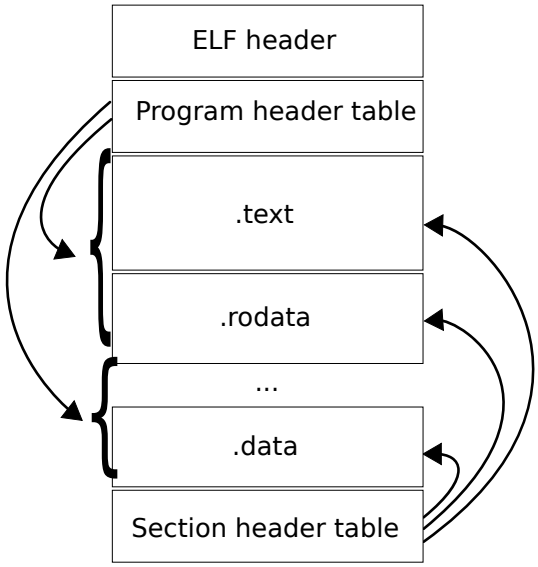


FIGURE 2 – Structure d'un fichier ELF. Source : Wikipedia.

Lorsque notre programme trouve une section de ce type, il récupère sa position, sa taille, alloue une région en mémoire et y copie les données. Il faut prendre soin de conserver l'adresse à laquelle les données doivent être chargées lors d'une véritable exécution, car celle-ci est différente de celle où on les charge dans le processus de l'analyseur. Lorsqu'une instruction analysée fera référence à une adresse donnée, on devra alors la translater de manière adéquate.

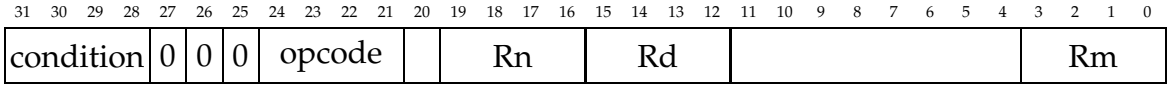
Table des symboles Pour une lecture plus agréable des graphes générés par l'analyseur, nous nommons les fonctions par leur véritable nom, lorsque celui-ci est disponible. Ceci est possible en lisant la table des symboles du binaires, qui correspond à la section de type `SHT_SYMTAB`. Cette section, normalement utile à l'édition de liens dynamique, contient des noms associés à certaines adresses de l'espace mémoire du processus : fonctions, sections, segments...

3.2.2 Désassembleur

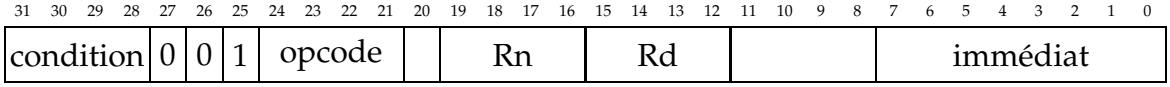
Le code de cette partie est présent dans le fichier `arm_instructions.c`. Elle est responsable de reconnaître les instructions de code machine (grâce aux informations du manuel de référence de l'architecture ARM [6]), et notamment les instructions de branchement. Celles-ci sont toutes celles qui modifient le registre PC (*Program Counter*).

Il en existe différent types, les plus simples étant les instructions de branchement explicites. Il s'agit de `b`, `bl`, `bx`, `blx` et `bxj`. En plus de celles-là, il faut prendre en compte toutes les opérations usuelles qui agissent sur le registre PC, par exemple avec une simple addition. Le nombre de ces instructions étant très élevé, nous avons préféré les classer par catégories. Nous représentons ces catégories plus bas.

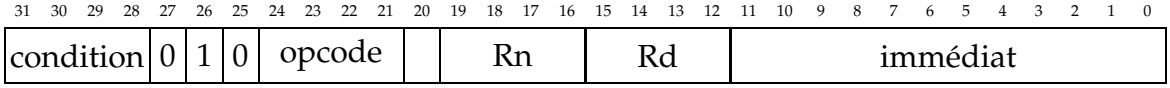
Opérations à décalage :



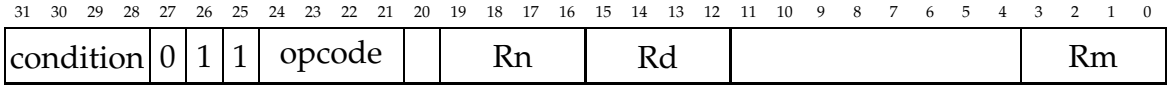
Opérations avec immédiat :



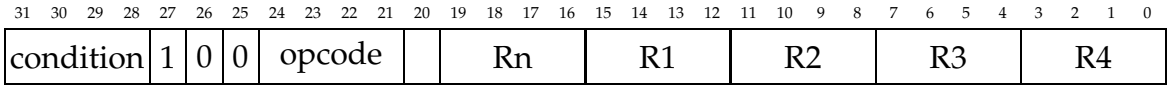
Load/store avec *offset* immédiat :



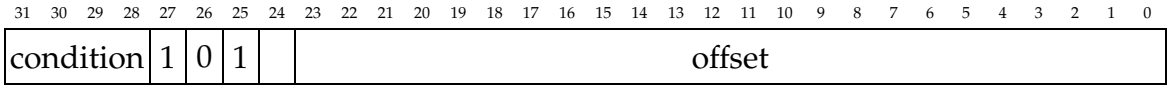
Load/store avec *offset* registre :



Load/store multiple :



Branchement et branchement avec lien (`b`, `bl`, `blx`) :



Pour tester si une instruction est un branchement, il faut donc traiter les cas précédents, en vérifiant si le registre PC est affecté par l'opération.

Cette partie calcule aussi l'adresse du branchement lorsque cela est possible, c'est à dire dans le cas où l'instruction est un branchement vers une adresse donnée par un *offset* : `b`, `bl` ou `blx`.

3.2.3 Décompilateur

Le code de cette partie est présent dans le fichier `decompiler.c`. La procédure de détection des fonctions se divise en trois grandes étapes :

1. Parcours de toutes les instructions du binaire, avec enregistrement de toutes les déclarations (*statements*) d'intérêt.

2. Découverte des fonctions, de leurs adresses de début et de fin, en utilisant les déclarations.
3. Détermination des appels, sauts et retour au sein des fonctions trouvées.
4. Détection des appels système au sein des fonctions.

Détection des déclarations Les déclarations, ou *statements*, représentent les points particuliers du code du binaire. Il peut s'agir :
- d'instructions de branchement (BRANCH),
- d'instructions vides (NOP),
- d'appels système,
- de valeurs stockées en dur (WORD).

Une déclaration peut être conditionnelle ou inconditionnelle. Parmi les branchements, on définit trois sous-types : les appels de fonctions (CALL), les sauts définitifs (JUMP) et les retours de fonctions (RETURN).

La détection consiste à lire toutes les instructions depuis un point de départ, et à s'arrêter lorsque l'on trouve une instruction de retour (RETURN) ou de saut (JUMP) inconditionnel.

Pendant cette lecture, si on a trouvé un branchement dont l'adresse est calculable (branchement statique), on ajoute l'adresse de destination à la liste des adresses à explorer. Ainsi, en sautant d'appel en appel, on parcourt les fonctions contenues dans l'exécutable. Cette procédure est décrite dans le pseudo-algorithme suivant.

```
1:  declarations ← []
2:  a_explorer ← []
3:  a_explorer.ajouter(point_d_entree)
4:  for all adresse_de_depart ∈ a_explorer do
5:      for (adresse ← adresse_de_depart ; ; adresse ← adresse + 4) do
6:          if deja_visitee(adresse) then
7:              sortir
8:          end if
9:          declarer_deja_visitee(adresse)
10:         instruction ← lire(adresse)
11:         if est_un_branchement(instruction) then
12:             declarations.ajouter(BRANCH, adresse)
13:             if adresse_de_destination_calculable(instruction) then
14:                 destination ← calculer_adresse(instruction)
15:                 a_explorer.ajouter(destination)
16:             end if
17:             if est_un_retour(instruction) then
18:                 sortir
19:             else if est_un_saut_inconditionnel(instruction) then
20:                 sortir
21:             end if
```

```
22:         else if est_un_nop(instruction) then
23:             declarations.ajouter(NOP, adresse)
24:         else if est_un_load_ou_store(instruction) then
25:             destination ← calculer_adresse(instruction)
26:             declarations.ajouter(WORD, destination)
27:         end if
28:     end for
29: end for
```

La figure 3 montre la détection des déclarations sur un exemple de programme.

Détection des fonctions Il faut savoir qu'il n'y a pas d'instruction d'appel de fonction en ARM, ni de retour. Les classiques `call` et `ret` du jeu d'instructions x86 n'existent pas dans celui d'ARM. À leur place, il existe un registre LR (*Link Register*), conçu pour stocker l'adresse de retour lors d'un appel de fonction.

Pour les appels de fonctions, toute la difficulté est de détecter les branchements qui sont effectivement des appels, en les différenciant des autres branchements qui proviennent de *if/else*. Voici comment nous procédons.

Nous considérons comme des appels :

- les instructions *branch and link*, qui sauvegardent la valeur du registre PC (*Program Counter*) dans le registre LR (afin de sauvegarder l'adresse à laquelle retourner à la fin de la procédure) ;
- les sauts inconditionnels vers une adresse hors de la fonction (ce qui signifie qu'on ne reviendra pas : la fonction est terminée).

Le problème est de définir « hors de la fonction » : si le saut inconditionnel est 8 octets plus loin, ce n'est pas un saut vers une autre fonction, mais probablement le saut qui permet d'éviter un bloc *else*. Pour pallier à cela, nous définissons une variable $fin\_minimum$, qui contient l'adresse minimale à laquelle la fonction peut s'arrêter. Cette valeur est mise à jour à chaque saut conditionnel (par exemple, bloc *if*), pour dire que la fin de la fonction est au minimum après le bloc *if*.

Pour plus de clarté, ce comportement est illustré dans les figures 4 et 5.

Pour ce qui est des retours, nous les caractérisons dans les deux cas suivants :

- les branchements dont la destination est l'adresse dans LR : `bx lr`, de code `0xe12fff1e`;
- les dépilements d'une valeur vers le registre PC (*Program Counter*) : `pop [fp, pc]`, de code `0xe8bd8800`.

Il est nécessaire de préciser que cette solution ne couvre pas tous les cas, car certains retours peuvent être exotiques et il n'existe pas de méthode universelle pour les détecter. Le problème est discuté plus en détail dans la section 3.4.

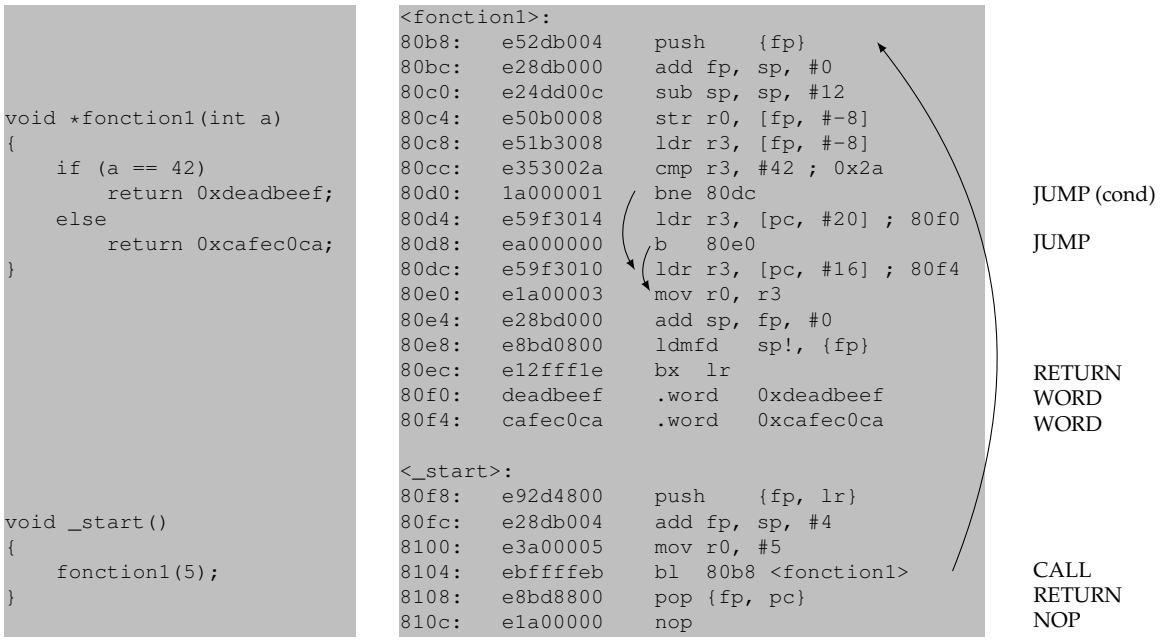


FIGURE 3 – Exemple de déclarations dans un programme simple. De gauche à droite : source C ; assembleur généré ; déclarations détectées par l'analyseur.

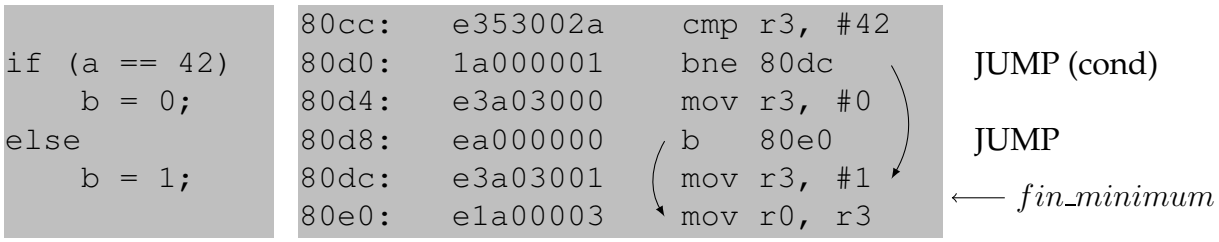


FIGURE 4 – Exemple de détection de fonction avec traitement de saut inconditionnel. L'instruction en `0x80d8` n'est pas considérée comme un appel à une nouvelle fonction car $fin\_minimum$ a été placée après lors du décodage du saut conditionnel en `0x80d0`.

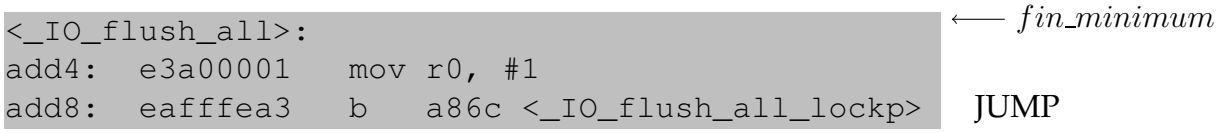
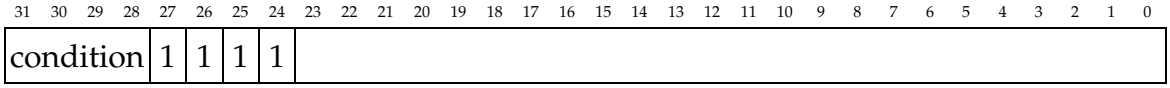


FIGURE 5 – Exemple provenant d'une fonction de la librairie C standard : `_IO_flush_all`. L'instruction en `0xadd8` est considérée comme un appel à une nouvelle fonction car $fin\_minimum$ pointe vers une adresse antérieure. La fonction `_IO_flush_all` se termine donc à cet endroit.

La procédure globale est synthétisée dans le pseudo-algorithme suivant.

```
1: tri_fusion(declarations)
2: fonctions ← []
3: fonctions.ajouter_fonction(start ← point_d_entree)
4: for all f ∈ fonctions do
5:     fin_minimum ← 0
6:     for all d ∈ declarations_apres_adresse(f.start) do
7:         if d.type = NOP or d.type = WORD then
8:             if fin_minimum ≤ d.adresse + 4 then
9:                 f.end ← d.adresse
10:                sortir
11:            end if
12:        else if d.type = RETURN then
13:            if fin_minimum ≤ d.adresse + 4 then
14:                f.end ← d.adresse + 4
15:                sortir
16:            end if
17:        else if d.type = JUMP and branchement_inconditionnel(d) then
18:            if fin_minimum ≤ d.adresse + 4 then
19:                f.end ← d.adresse + 4
20:                if adresse_de_destination_calculable(d) then
21:                    destination ← calculer_adresse(d)
22:                    if destination < f.start or destination ≥ d.adresse + 4 then
23:                        fonctions.ajouter_fonction(start ← destination)
24:                    end if
25:                end if
26:                sortir
27:            end if
28:        else if d.type = JUMP and adresse_de_destination_calculable(d) then
29:            destination ← calculer_adresse(d)
30:            if destination + 4 < fin_minimum then
31:                fin_minimum ← destination + 4
32:            end if
33:        else if d.type = CALL and adresse_de_destination_calculable(d) then
34:            destination ← calculer_adresse(d)
35:            fonctions.ajouter_fonction(start ← destination)
36:        end if
37:    end for
38: end for
```

Détection des appels système Finalement, l'analyseur effectue une dernière passe pour détecter les appels système. Ceux-ci sont caractérisés par des instructions du type de celle représentée plus bas. Sous GNU/Linux, le numéro de l'appel doit être stockée dans le registre 7, on peut donc trouver ce numéro dans l'instruction précédant l'appel système. On aurait par exemple pour un open (appel numéro 5), l'instruction `mov r7, #5`, suivie de l'appel système représenté ici.



3.3 Génération de graphes

3.3.1 Graphe d'appel

À partir des informations collectées, le graphe d'appel est assez simple à construire. Rappelons qu'après analyse du binaire, nous avons en mémoire :

- la liste des fonctions ;
- la liste des déclarations (sauts, appels, retours, instructions WORD et NOP, appels système) de chaque fonction.

Pour produire le graphe d'appel dans un format interprétable par GraphViz, il faut afficher :

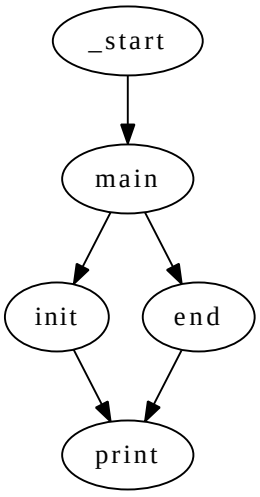
- un ensemble de sommets qui correspondent aux fonctions ;
- un ensemble d'arêtes qui représentent les appels d'une fonction vers une autre.

C'est exactement ce que nous faisons : pour chaque fonction nous affichons son nom et ses instructions de branchement qui sont des appels vers d'autres fonctions.

Voici un exemple de programme en C, la sortie produite par l'analyseur, et le graphe généré par GraphViz.

```
void print(char *str)
{
    // do something
}
void init()
{
    print("init...");
}
void end()
{
    print("end...");
}
void main()
{
    init();
    end();
}
void _start()
{
    main();
}
```

```
digraph G {
    F0 [label="_start"];
    F0 -> F1;
    F1 [label="main"];
    F1 -> F2;
    F1 -> F3;
    F2 [label="init"];
    F2 -> F4;
    F3 [label="end"];
    F3 -> F4;
    F4 [label="print"];
}
```



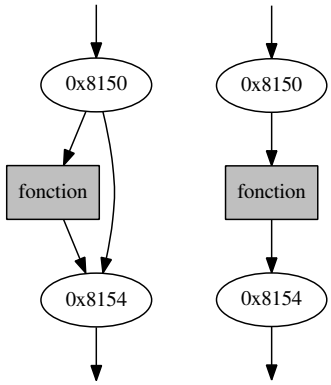
3.3.2 Graphe de flot de contrôle intra-procédural

Le graphe de flot de contrôle (CFG) d'un programme montre l'ensemble des chemins qui peuvent être suivis durant l'exécution. Il met notamment en évidence les branchements conditionnels liés aux tests et aux boucles. Nous ne souhaitons pas produire de CFG global représentant tout l'exécutable, car cela ferait un graphe énorme et difficilement lisible. Au lieu de cela, nous préférerons générer un CFG intra-procédural, montrant le flot de contrôle au sein d'une fonction particulière choisie par l'utilisateur.

Rappelons qu'avant de générer le graphe de flot de contrôle d'une fonction, nous avons en mémoire la liste de tous ses branchements, ainsi que leurs caractéristiques. Nous utilisons ces informations pour produire le graphe, en respectant les principes suivants :

Points d'entrée et de sortie Nous créons deux nœuds spéciaux dans le graphe, correspondant aux adresses de début et de fin de la fonction analysée. Le nœud de début est connecté à la première instruction ; tandis que toutes les instructions de retour plus la dernière instruction sont connectées au nœud de fin.

Appels Un appel de fonction (CALL) donnera naissance à trois nœuds dans le graphe : celui de l'adresse de l'instruction appelante, celui de la fonction et celui de l'adresse de retour (adresse appelante + 4 octets). Le flux de contrôle passe successivement par ces trois nœuds, dans l'ordre. Dans le cas d'un appel avec condition (instruction conditionnelle), on ajoute une arête allant directement du premier au troisième nœud. Celle-ci représente le cas où la condition n'est pas vérifiée, et l'instruction d'appel non exécutée.

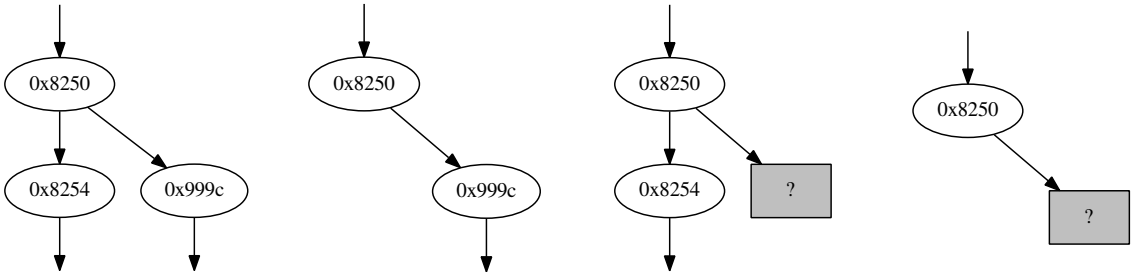


Le mécanisme que nous venons de décrire est justifié par le fait qu'un appel de fonction est sensé être suivi d'un retour, qui ramène le flot d'exécution juste après l'instruction appelante (adresse + 4 octets). Cela n'est pas toujours le cas, car certaines fonctions ne retournent pas. Ce problème est dû à l'optimisation de la compilation, qui est discutée dans la section 3.4.3.

Pour les appels système (SYSCALL), le principe est le même : on crée trois nœuds et deux ou trois arêtes.

Sauts On considère deux types de sauts (JUMP) :

- vers une adresse connue à l'intérieur de la fonction analysée ;
- vers une adresse connue à l'extérieur de la fonction, ou inconnue (calculée dynamiquement).



Pour chacun de ces deux types, le saut peut être conditionnel ou non. Lorsqu'il l'est, on ajoute une arête vers le nœud représentant l'instruction à l'adresse + 4 octets. Ceci correspond à la non-exécution du saut, due à une condition non vérifiée.

Pour les sauts vers une adresse connue à l'intérieur de la fonction, on ajoute simplement une arête vers le nœud de l'adresse correspondante. Pour les autres, on crée un nœud spécial, inconnu, vers lequel est transféré le flot d'exécution.

Retours Pour les retours, on trace simplement une arête allant du nœud de l'instruction jusqu'au nœud représentant la fin de la fonction.

L'algorithme conçu pour construire le CFG intra-procédural se divise en six étapes. Elles sont toutes nécessaires à la création du graphe. Des essais ont été faits pour grouper des opérations et accélérer le calcul, mais cela a généré des problèmes pour certains cas particuliers. Pour cette raison, nous gardons ces six étapes, qui décomposent le travail de manière claire :

1. Détection de toutes les adresses utiles à la construction du graphe. Ceci est fait en utilisant les déclarations enregistrées pendant la phase de décompilation (voir section 3.2.3). On enregistre notamment : les adresses des branchements, les adresses les suivant directement (+ 4 octets) s'ils sont conditionnels, les adresses de destination des branchements.
2. Tri croissant des adresses, et suppression des doublons.
3. Établissement des correspondances entre les nœuds et les déclarations.
4. Traçage des arêtes partant de chaque nœud, en utilisant les déclarations associées.
5. Simplification : suppression des nœuds inutiles. La détection de l'étape 1 a généré plus de nœuds que nécessaire, nous retirons donc ceux qui n'ont qu'un seul parent et un seul enfant, et qui sont des nœuds réguliers (pas des appels de fonction ou des appels système).
6. Génération du descripteur de graphe.

La sortie de cet algorithme, au format texte, est un descripteur comprenant les nœuds et les arêtes. Elle peut être utilisée pour dessiner un graphe avec GraphViz.

Voici un exemple de programme en C, la sortie produite par l'analyseur, et le graphe généré par GraphViz.

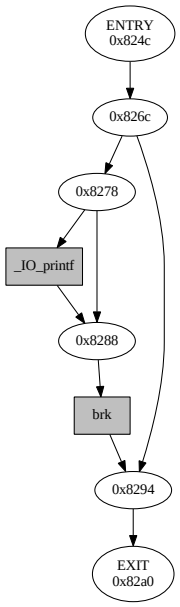
```c
#include <stdio.h>

void main(int argc)
{
  if (argc == 0)
    return;

  else if (argc > 0)
    printf("Ah.");

  brk();
}
```

```
digraph G {
    N_0_824c [label="ENTRY\n0x824c"];
    N_0_824c -> N_0_826c;
    N_0_826c [label="0x826c"];
    N_0_826c -> N_0_8278;
    N_0_826c -> N_0_8294;
    N_0_8278 [label="0x8278"];
    N_0_8278 -> N_1_8284;
    N_0_8278 -> N_0_8288;
    N_1_8284 [label="_IO_printf"];
    N_1_8284 -> N_0_8288;
    N_0_8288 [label="0x8288"];
    N_0_8288 -> N_1_8288;
    N_1_8288 [label="brk"];
    N_1_8288 -> N_0_8294;
    N_0_8294 [label="0x8294"];
    N_0_8294 -> N_0_82a0;
    N_0_82a0 [label="EXIT\n0x82a0"];
}
```



3.4 Difficultés rencontrées

3.4.1 Branchements dynamiques

Un branchement est dynamique quand l'adresse vers laquelle aller est calculée, et non indiquée statiquement dans le programme. Une adresse calculée dépend potentiellement d'une entrée utilisateur, d'une variable d'environnement ou de l'état du système d'exploitation, on ne peut donc pas la prévoir par notre analyse statique, en général. Les risques avec les sauts dynamiques sont, entre autres, de passer à côté d'un appel de fonction sans savoir comment y aller, ou bien de ne pas détecter un retour et de penser que la fonction continue plus bas.

Cette difficulté n'a pas de solution générale, mais pourrait être réduite en utilisant des techniques d'analyse dynamique. Ceci pourrait être fait dans de futurs travaux.

3.4.2 Mauvaise interprétation

Certaines instructions ou suites d'instructions peuvent être mal interprétées. Voici quelques exemples.

1. Retour non détecté. Imaginons un retour de fonction, qui peut par exemple s'exprimer `bx lr` (branchement vers l'adresse de retour, stockée dans le registre LR). Si cette instruction est remplacée par les deux suivantes : `mov r3, lr` et `bx r3`, le retour ne sera pas détecté.
2. Faux retour. À l'inverse, si on a `mov lr, #42` suivi de `bx lr`, l'analyseur croira à un retour alors qu'il s'agit d'un saut vers l'adresse `42`.
3. Branchement vers une adresse *a priori* dynamique. Un appel vers la fonction à l'adresse `0x42` s'écrirait normalement `bl 42`. Encore une fois, si l'on

a `mov r1, #0x42` suivi de `bl r1`, l'analyseur croira à un branchement vers l'adresse contenue dans r1. Cette adresse n'étant *a priori* pas déterminable, il enregistrera un saut dynamique et n'ira pas explorer la fonction à l'adresse `0x42`.

On pourrait penser qu'une solution simple serait de lire les instructions précédant un branchement *a priori* dynamique, pour voir si l'adresse de destination est donnée juste avant. Malheureusement, on ne peut pas se fier aux instructions précédant le branchement, car il pourrait très bien y avoir un saut depuis une adresse lointaine vers l'instruction de branchement, sans passer par les instructions situées juste avant celle-ci.

En revanche, nous pensons que la génération du graphe des dominateurs pourrait pallier à ce problème. Ceci pourrait être fait dans de futurs travaux.

3.4.3 Optimisations du compilateur

Lorsqu'un programme est optimisé, le compilateur réordonne les instructions pour réduire la taille du binaire, et rendre son exécution plus rapide. Dans certains cas d'optimisations poussées, le réarrangement du binaire complique notre tâche car il peut empêcher de détecter des appels ou des retours de fonctions. Lorsqu'il optimise, le compilateur peut, entre autres :

- supprimer des fonctions simples pour les inclure directement dans le code de la fonction appelante ;
- imbriquer des fonctions entre elles ;
- rendre des appels de fonction dynamiques.

Dans un premier temps, nous avons travaillé sur des binaires compilés avec l'option `-O0` de gcc, qui désactive l'optimisation. Depuis, nous avons étudié les réarrangements commis par l'optimisation, afin de la prendre en compte dans l'analyse du binaire. Nous sommes maintenant capables d'analyser des binaires avec l'optimisation par défaut (`-O2`). Cependant, nous n'avons pas étudié les effet d'une optimisation plus poussée. Par conséquent, l'analyseur peut rencontrer des problèmes s'il essaie de reconstruire des programmes compilés avec `-O3`.

3.4.4 Fonctions imbriquées

Dans certaines fonctions optimisées, notamment celles de la librairie C standard, l'analyseur ne détecte pas d'instruction de retour. Le contrôle passe donc à la fonction suivante, et la procédure se termine à la fin de cette dernière. Dans ce cas, on obtient deux fonctions imbriquées, ayant la même adresse de fin mais des débuts différents.

```
<fonction_complete>:
a000:  e3a00001   mov r0, #1
        ...
        bloc 1
        ...
```

```
a03c:  e3a00005   mov r0, #5        ; pas de retour
<sous_fonction>:
a040:  e1a03000   mov r3, r0
        ...
        bloc 2
        ...
a05c:  e12fff1e   bx lr     ; retour commun aux 2 fonctions
```

La solution retenue pour gérer ces situations est d'effectuer une passe pour trouver les fonctions imbriquées. À chaque couple de fonctions trouvé, on change l'adresse de fin de la première pour qu'elle corresponde au début de la seconde, et ajoute un appel à la deuxième à la fin de son code.

Ceci est fait par la fonction `rp_fix_overlapping_functions` de `decompiler.c`.

3.4.5 La librairie C

Les logiciels sont souvent compilés en incluant la librairie C standard, qui fournit de nombreuses fonctionnalités usuelles (flux `stdin`, fonctions `fopen`, `strncpy`) et encapsule les appels systèmes. Ceci a plusieurs conséquences néfastes sur le fonctionnement de notre analyseur ARM :

- la libc ajoute plus de 800 fonctions au programme ;
- ces fonctions sont optimisées, imbriquées et à sauts dynamiques ;
- le point d'entrée du programme (`_start`) n'appelle pas la fonction `main`, mais `__libc_start_main`, et l'appel à la fonction `main` est dynamique donc *a priori* pas détectable par l'analyseur.

Pour cette raison, nous avons commencé notre étude avec des binaires compilés avec l'option `-nostdlib`. Depuis la première version, nous avons fait les modifications nécessaires pour que la librairie C standard soit prise en charge.

Premièrement, nous avons remarqué que l'adresse de la fonction `main` est toujours stockée dans un WORD à l'adresse virtuelle `0x8184`. C'est la fonction `_start` qui se charge de lire ce WORD et de le passer en argument à `__libc_start_main`. Cette dernière appelle `main` en faisant un branchement dynamique vers cette adresse. Ce branchement est toujours situé à l'adresse de `__libc_start_main` + `0x1a8`.

Notre solution comporte donc les étapes suivantes :

- test de présence de la libc ;
- le cas échéant, remplacement du branchement dynamique en (`__libc_start_main` + `0x1a8`) par un appel statique vers l'adresse contenue dans le WORD en `0x8184`.

D'autre part, le nombre de fonctions ajoutées par la librairie C est très important. Même si elles ne sont pas toutes détectées par l'analyseur (cela dépend de si elles sont appelées ou pas), il en reste plusieurs centaines. Il est nécessaire de pouvoir masquer ces fonctions. Pour cela, nous les marquons avec un drapeau et proposons une option pour les omettre lors de l'affichage des graphes.

3.5 Performances

Même si ce programme est un projet de démonstration, nous pensons qu'il est important d'analyser ses performances pour comprendre les points critiques et voir quels choix algorithmiques affectent le temps d'exécution.

Optimisation des boucles Nous avons utilisé les outils perf [4] et valgrind [5] pour obtenir des statistiques sur l'exécution de notre programme.

```
perf stat ./arm-analyser
```

```
valgrind --tool=callgrind ./arm-analyser
```

La première commande permet d'avoir des informations générales : temps d'exécution total, temps passé en attente du système d'exploitation, nombre de fautes de page... La seconde procède à une étude plus approfondie pour donner le temps total passé dans chaque fonction.

Ces outils nous ont permis de détecter et de corriger une mauvaise implémentation qui faisait passer 80% du temps dans la fonction `memcmp`. Le problème était le suivant. Tous les tableaux utilisés par l'analyseur sont à longueur dynamique, pour cela ils sont gérés par des macros définies dans `common.h`. Dans la première implémentation, la longueur d'un tableau n'était stockée nulle part, et la fin de celui-ci était marqué par un élément nul. La macro d'itération sur ces tableaux testait donc, pour chaque élément, s'il était nul (avec `memcmp`) et dans l'affirmative, s'arrêtait. Vue la longueur ce certains tableaux (la liste des déclarations par exemple), le parcours était fortement ralenti par la comparaison systématique de ses éléments à l'élément nul. Depuis, nous avons modifié la façon dont les tableaux sont gérés : ceux-ci sont toujours à longueur dynamique, mais leur longueur est enregistrée à l'adresse précédant le début du tableau, ce qui nous affranchit du test avec `memcmp`. Le temps d'exécution a été divisé par quatre.

Parallélisation Il est possible d'améliorer la vitesse d'exécution de notre programme en tirant profit de la parallélisation des fils d'exécution (*multi-threading*). Pour cela, il faut déterminer quelles portions des algorithmes n'ont pas de dépendances vis-à-vis des autres. Les portions indépendantes peuvent alors s'exécuter en parallèle sans affecter le résultat.

Par exemple, on pourrait diviser la partie qui parcourt le programme et lit les instructions en quatre *threads*, chacun étant chargé d'une partie du binaire. De la même façon, la reconnaissance des fonctions en utilisant les déclarations peut être divisée en plusieurs sous-tâches travaillant chacune à des adresses différentes. Ceci pourra être fait dans de futurs travaux.

date	étape	
11 février	Avoir un programme capable de charger des binaires, charger leurs sections exécutables et trouver le point d'entrée.	✓
	Créer un binaire de test assez simple, compilé pour ARM.	✓
	Commenter le code et documenter le projet.	✓
	Gérer les programmes compilés sans libc ni optimisation (`-O0`).	✓
	Afficher la liste des fonctions et leurs adresses.	✓
18 mars	Créer des binaires ARM réalistes et plus compliqués : la suite Coreutils (ls, cat, echo, wc...)	✓
	Améliorer la méthode de détection des fonctions en faisant une liste de toutes les instructions de branchement.	✓
	Résoudre le problème des fonctions imbriquées.	✓
	Trouver les noms des fonctions dans la table des symboles.	✓
	Gérer les programmes compilés avec la libc et `-O0`.	✓
	Afficher le graphe d'appel.	✓
15 avril	Trouver une solution pour le problème des sauts dynamiques.	
	Gérer certaines parties de la librairie C standard.	✓
	Gérer les programmes compilés avec la libc et `-O2`.	✓
	Analyser les performances pour optimiser le temps d'exécution.	✓
	Avoir une analyse de vrais programmes (true, wc, echo) qui est fidèle à la réalité.	✓
	Afficher le graphe de flot de contrôle intra-procédural.	✓

TABLE 1 – Échéancier

4 Résultats

4.1 Utilisation du livrable

4.1.1 Compilation

Dépendance Pour compiler le projet, il convient de s'assurer d'avoir la dépendance requise : la libelf. Cette librairie s'installe avec la paquet `libelf-dev` sous Debian, ou `elfutils-libelf-devel` sous Fedora.

Compilation La compilation s'effectue ensuite avec la commande `make`, ce qui exécute la commande :

```
gcc -std=gnu11 -Wall -o arm-analyser -lelf *.c *.h
```

et génère notre analyseur ARM : `arm-analyser`.

4.1.2 Analyse

L'exécution de l'analyseur sans argument affiche l'aide :

```
Usage: ./arm-analyser action [options] program
actions:
  help      display this help
  fn        dump functions
  cg        generate callgraph
  cfg       generate CFG (option -f needed)
options:
  -s        show standard C library
  -f FN     limit action to function FN (name or address)
options for action fn:
  -c        compact dump (names, addresses and childs)
  -cc       very compact dump (only addresses)
```

Affichage des fonctions Pour lancer l'analyse du binaire et afficher toutes les fonctions détectées, il faut utiliser l'action `fn`. L'option `-c` permet un affichage condensé, plus facile à lire, et l'option `-s` affiche aussi les fonction de la librairie C (qui sont masquées par défaut, voir la section 3.4.5). On obtient une liste des fonctions, avec leurs adresses de début et de fin, ainsi que les fonctions qu'elles appellent. Exemple :

```
$ ./arm-analyser fn -c programme
main        0x00008388  0x000083d0  fonction1,fonction2
fonction1   0x000082e0  0x0000833c
fonction2   0x0000833c  0x00008388  fonction3
fonction3   0x0000824c  0x00008274
```

Affichage des déclarations Lors de l'affichage des fonctions, si l'on omet l'option `-c`, toutes les déclarations sont affichées. En combinant l'affichage avec l'option `-f`, qui n'affiche que la fonction désirée, on obtient toutes les déclarations détectées au sein de la fonction. Exemple :

```
$ ./arm-analyser fn -f main programme
main
  08388 {
  083ac   BRANCH ( CALL )   static addr -> 082e0 (fonction1)
  083bc   BRANCH ( CALL )   static addr -> 0833c (fonction2)
  083cc   BRANCH (RETURN)   dynam. addr
  083d0 }
```

Génération du graphe d'appel Pour générer un descripteur de graphe d'appel utilisable par GraphViz, on utilise l'action `cg`. Exemple :

```
$ ./arm-analyser cg programme
```

Génération du graphe de flot de contrôle Pour générer un descripteur de CFG pour une fonction en particulier, on utilise l'action `cfg` avec l'option `-fn`. Exemple :

```
$ ./arm-analyser cfg -f fonction1 programme
```

4.1.3 Affichage des graphes

Une fois que l'analyseur a produit un descripteur de graphe, il faut le tracer avec GraphViz. Ce logiciel peut s'utiliser avec différentes commandes, selon la géographie voulue pour le graphe. Les plus communs sont `dot` (affichage hiérarchique, de haut en bas) et `neato` (affichage centré).

Exemple de traçage du graphe d'appel de l'exécutable `programme` dans le fichier `graphe.eps` :

```
$ ./arm-analyser cg programme | dot -T eps -o graphe.eps
```

4.2 Exemples

Nous présentons ici des exemples d'analyses de binaires, montrant les résultats de notre projet.

Tous les résultats donnés dans cette section peuvent être reproduits en utilisant les programmes de démonstration fournis dans l'archive. Ces programmes sont situés dans le dossier `test`.

4.2.1 helloworld

Ce programme très simple vise à faire une première démonstration de notre analyseur ARM.

Voici son code source :

```
int one_arg(int val)
{
    return val + 1;
}
int two_args(int val1, int val2)
{
    return val1 + val2;
}
int three_args(int val1, int val2, int val3)
{
    return val1 * val2 - val3;
}
int test(int val)
{
    if (val > 5)
        return one_arg(val);
    else if (val < 0)
        return two_args(42, val);
    return 9;
}
```

```
void operations(int val)
{
    one_arg(val++);
    two_args(42, val);
    three_args(87, 88, 89);
}
int main(int argc, char **argv)
{
    int a = test(argv[0][0]);
    operations(a);
    return a;
}
```

Il est compilé avec gcc, en incluant la libc :

```
$ arm-linux-gnueabi-gcc -Wall -O0 -static -march=armv5 \
    -o helloworld helloworld.c
```

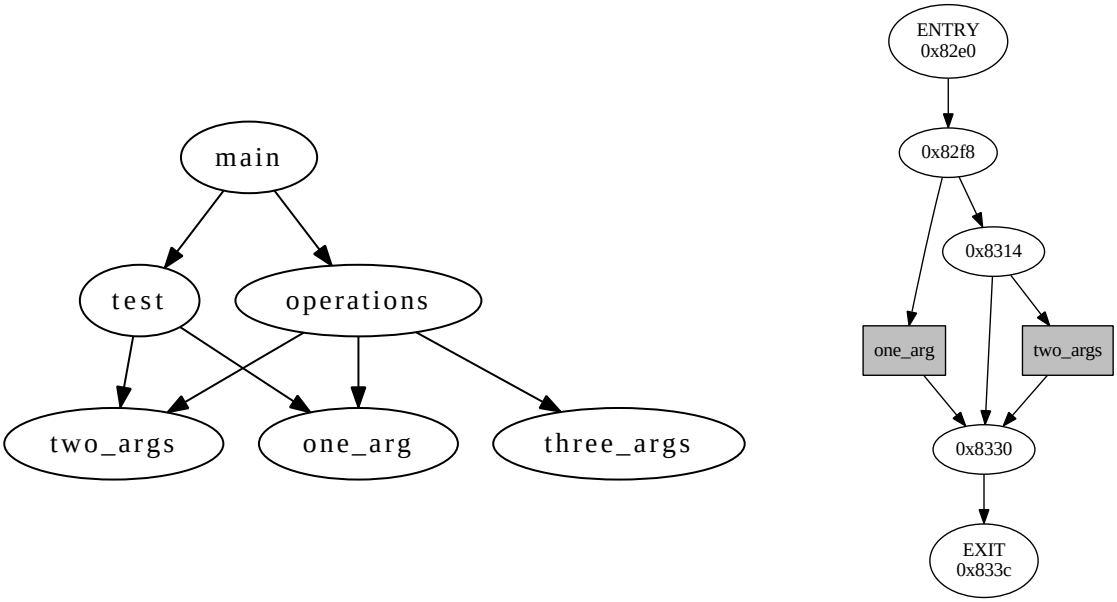
Résultat de l'analyse avec l'action `fn` et en cachant la libc :

```
$ ./arm-analyser fn test/helloworld
main
    08388 {
    083ac   BRANCH ( CALL )         static addr  -> 082e0 (test)
    083bc   BRANCH ( CALL )         static addr  -> 0833c (operations)
    083cc   BRANCH (RETURN)         dynam. addr
    083d0 }
test
    082e0 {
    082f8   BRANCH ( JUMP )  cond.  static addr  -> 0830c
    08300   BRANCH ( CALL )         static addr  -> 0824c (one_arg)
    08308   BRANCH ( JUMP )         static addr  -> 08330
    08314   BRANCH ( JUMP )  cond.  static addr  -> 0832c
    08320   BRANCH ( CALL )         static addr  -> 08274 (two_args)
    08328   BRANCH ( JUMP )         static addr  -> 08330
    08338   BRANCH (RETURN)         dynam. addr
    0833c }
operations
    0833c {
    08360   BRANCH ( CALL )         static addr  -> 0824c (one_arg)
    0836c   BRANCH ( CALL )         static addr  -> 08274 (two_args)
    0837c   BRANCH ( CALL )         static addr  -> 082a4 (three_args)
    08384   BRANCH (RETURN)         dynam. addr
    08388 }
one_arg
    0824c {
    08270   BRANCH (RETURN)         dynam. addr
    08274 }
two_args
    08274 {
```

```
    082a0   BRANCH (RETURN)         dynam. addr
    082a4 }
three_args
    082a4 {
    082dc   BRANCH (RETURN)         dynam. addr
    082e0 }
```

Résultat de l'analyse, avec traçage du graphe d'appel et du CFG de la fonction `test` :

```
$ ./arm-analyser cg test/helloworld | dot -Teps -o callgraph.eps
$ ./arm-analyser cfg -f test test/helloworld | dot -Teps -o cfg-test.eps
```



4.2.2 GNU Coreutils

Afin de tester l'analyseur ARM sur de véritables programmes, nous fournissons dans l'archive certains binaires de la suite GNU Coreutils, déjà compilés pour l'architecture ARM. Il s'agit de `basename`, `cat`, `chmod`, `cp`, `echo`, `false`, `head`, `ls`, `md5sum`, `mv`, `pwd`, `rm`, `seq`, `sleep`, `true`, `wc` et `yes`.

La figure 6 montre le résultat de l'analyse sur le programme `sleep`. Ces graphes ont été obtenus avec les commandes :

```
$ ./arm-analyser cg test/coreutils/sleep | dot -Teps \
    -Gratio=fill -Gsize=3.5,10 -Grankdir=LR -o cg.eps
$ ./arm-analyser cfg -f main test/coreutils/sleep | dot \
    -Teps -o cfg.eps
```

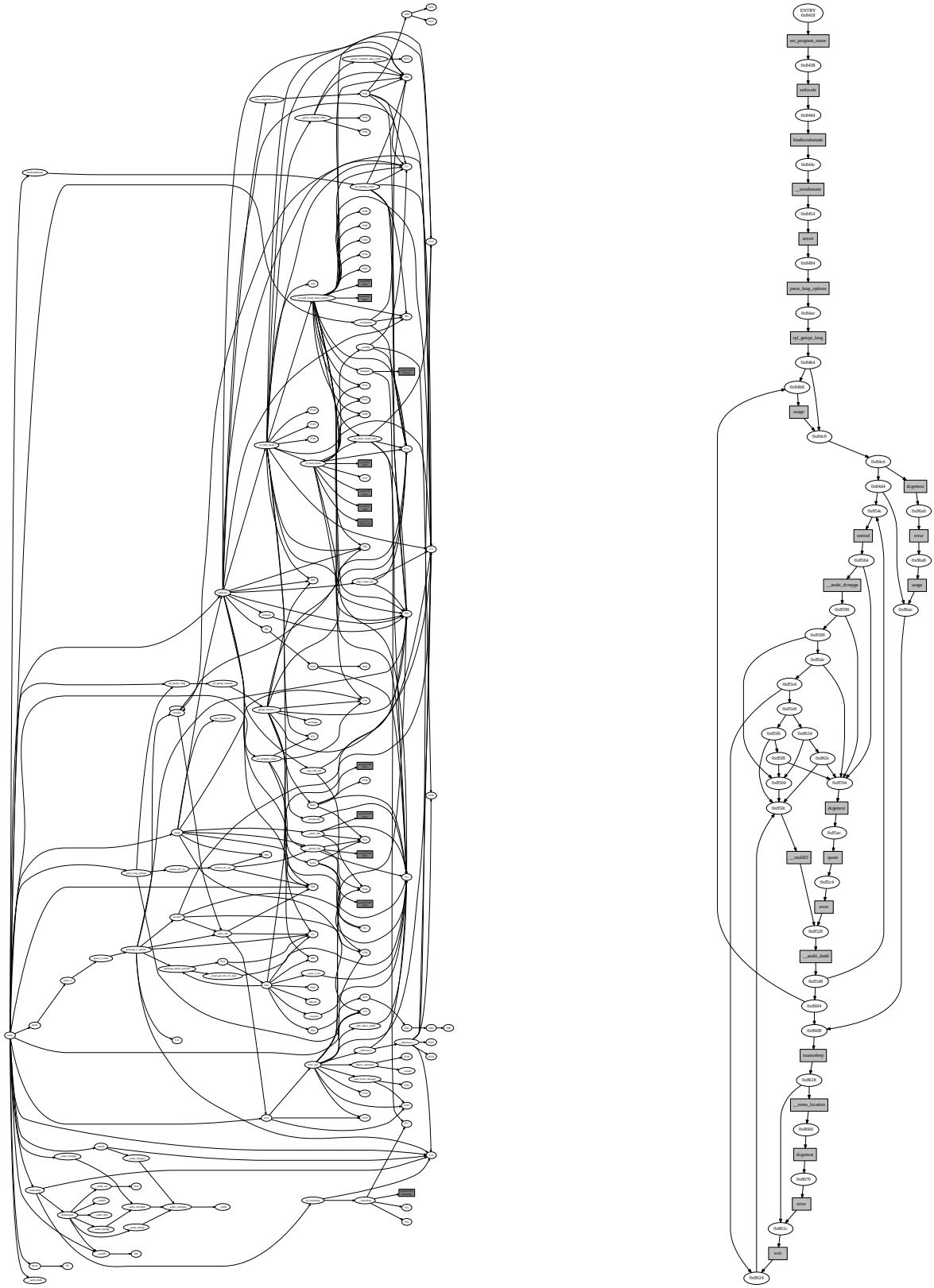


FIGURE 6 – Programme `sleep` : graphe d'appel et CFG de la fonction `main`

5 Discussion

Au cours de ce projet, nous avons étudié la structure d'exécutables binaires pour l'architecture ARM, et conçu un outil permettant d'extraire et d'afficher cette structure de manière automatique. Notre outil permet, entre autres, de générer le graphe d'appel du programme, et le graphe de flot de contrôle de ses fonctions. Cette analyse peut être à la base de l'étude complète d'un fichier binaire, telle qu'une décompilation.

Cependant, la portée de nos résultats est restreinte par les hypothèses que nous avons posées (voir la section 1.4). Il faut notamment souligner que les seuls binaires étudiés étaient au format ELF (systèmes GNU/Linux), compilés par `gcc`. Cela exclut par exemple, le portage de pilotes Windows (autorisé par le fabricant), mentionné dans la section 1.2.

Néanmoins, notre étude a posé des bases théoriques générales pouvant être réutilisées. Elle laisse la porte ouverte a une généralisation du processus, de manière à être compatible avec d'autres systèmes.

Références

[1] Boomerang. http://boomerang.sourceforge.net/. A general, open source, retargetable decompiler of machine code programs.

[2] Graphviz. http://www.graphviz.org/. Générateur de graphes.

[3] LTTng, the Linux Trace Toolkit next-generation. http://lttng.org/. Suite de traçage logiciel à haute performance.

[4] Perf. https://perf.wiki.kernel.org/. Profilage sur Linux et compteurs de performance.

[5] Valgrind. http://valgrind.org/. Suite d'outils d'analyse dynamique et d'instrumentation.

[6] ARM limited. *ARM Architecture Reference Manual*, 2005.

[7] Législation Canadienne. Loi sur le droit d'auteur. http://lois-laws.justice.gc.ca/fra/lois/C-42/TexteComplet.html, 1985-2012.

[8] Cristina Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, 1994.

[9] Free Software Foundation. GNU Binutils objdump. http://www.gnu.org/software/binutils/.

[10] Free Software Foundation. GNU Coreutils. http://www.gnu.org/software/coreutils/.

[11] Free Software Foundation. GNU Debugger (gdb). http://www.gnu.org/software/gdb/.

[12] Free Software Foundation. GNU Embedded Application Binary Interface (EABI) gcc. http://gcc.gnu.org/.

[13] Free Software Foundation. Librairie Libelf. http://directory.fsf.org/libs/misc/libelf.html.

[14] Hex-Rays. IDA, the Interactive Disassembler. https://www.hex-rays.com/products/ida/index.shtml.

[15] Microsoft. Windows Debugger (WinDbg). http://msdn.microsoft.com/en-us/windows/hardware/gg463009.aspx.

[16] TIS Committee. *Tool Interface Standard (TIS) Portable Formats Specification, version 1.2*, 1995.

[17] Oleh Yuschuk. OllyDbg. http://www.ollydbg.de/.