UNIVERSITÉ DU QUÉBEC À MONTRÉAL

CORRECTION AUTOMATIQUE DE TRAVAUX BINAIRES : DE LA DÉCOUVERTE DE CODE À LA GÉNÉRATION DE TESTS

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

PHILIPPE PÉPOS PETITCLERC

JANVIER 2019

TABLE DES MATIÈRES

[Page manquante]

LISTE DES FIGURES

LISTE DES TABLEAUX

RÉSUMÉ

Dans la foulée des événements entourant le *Cyber Grand Challenge*, organisé par le DARPA, un regain d'énergie s'est fait sentir dans le domaine de l'analyse de programmes binaires. De nouvelles techniques ont fait surface lors de la préparation des équipes pour cette compétition d'exploitation automatique et un effort remarquable d'implémentation a permis l'émergence de nouvelles stratégies et de nouveaux outils d'analyse de programmes binaires.

Dans cette étude, nous étudions différentes techniques d'analyse de programmes. Nous explorons comment les techniques traditionnelles se transposent vers l'analyse de programmes binaires et comment surmonter les complexités additionnelles que ces programmes présentent. Nous mettons l'accent sur les techniques modernes d'exécution symbolique de programmes binaires. Nous nous penchons également sur les méthodes de génération automatique de tests. En se basant sur ces recherches, nous proposons une stratégie de génération de suites de tests couvrantes sans accès au code source du programme testé ni à sa spécification.

Nous validons l'efficacité de la stratégie proposée en l'implémentant dans l'optique de la correction automatique de travaux universitaires soumis par des étudiants. Nous analysons ensuite la performance de la solution au travers plusieurs métriques récoltées lors de la correction de travaux. Nous enrichissons ses métriques avec de l'information extraites du code source assembleur des programmes.

MOTS-CLÉS : Analyse de programme, programmes binaires, génération de tests, exécution symbolique

INTRODUCTION

Contexte de l'étude

Analyser des programmes binaires, chercher des vulnérabilités logicielles, générer des exploits et les correctifs, le tout de manière complètement autonome. Voilà ce que devaient accomplir les systèmes de raisonnement automatique qui participaient au *Cyber Grand Challenge* organisé par le *DARPA*. Avec cet événement, un regain d'intérêt se fait sentir dans le domaine de l'analyse de programmes binaires et de nouvelles technologies émergentes évoquent un futur prometteur pour ce champ de recherche.

Une question se pose : ces nouvelles techniques de pointe en analyse de programmes binaires sont-elles utiles dans le cadre de la correction automatique de travaux pratiques en informatique ? En particulier, le cours INF2170, *Organisation des ordinateurs et assembleur*, du programme d'informatique et génie logiciel de l'Université du Québec à Montréal présente à ses étudiants le fonctionnement d'un ordinateur au travers *PEP/8*, un simulateur de processeur. Les étudiants y découvrent, entre autres, le lien entre le processeur, le code binaire, l'assembleur et le langage d'assemblage.

Nous nous sommes demandé s'il était possible d'automatiser une partie de la tâche de correction des travaux pratiques de ce cours. Nous cherchons donc à effectuer l'entièreté de la validation fonctionnelle des travaux pratiques de façon entièrement autonome.

Contribution scientifique

Cette étude présente donc une stratégie d'application des technologies de pointe en analyse de programmes binaires afin d'automatiquement identifier les comportements erronés dans les travaux soumis par les étudiants dans le cadre du cours INF2170. La stratégie profite de l'existence d'un programme de référence, une solution fonctionnelle écrite par l'enseignant, pour identifier les comportements déviants dans les soumissions des étudiants. En analysant directement le code binaire de référence et des soumissions et en raisonnant sur leur structure de code plutôt qu'en cherchant à inférer une spécification, nous sommes en mesure d'identifier les comportements incorrects des travaux étudiants. Cette approche permet également de générer automatiquement des cas de test (exemples d'entrées au programme) qui déclencheront les comportements erronés identifiés. Les étudiants pourront donc vérifier ces comportements incorrects en exécutant leur programme avec les entrées générées.

Nous présentons également un exemple d'implémentation de cette solution au travers d'une extension à ANGR, une suite d'analyse de programmes binaires, qui a permis d'évaluer l'intérêt de la stratégie proposée.

Organisation du document

Le chapitre 1 introduit le simulateur PEP/8 : la machine virtuelle pédagogique, l'architecture du processeur simulé et le langage d'assemblage associé. Nous y détaillons également son contexte d'utilisation à l'Université du Québec à Montréal.

Le chapitre 2 fait une brève introduction à l'analyse de programmes. Nous y présentons différents aspects de l'analyse de programmes. Nous y introduisons aussi l'analyse de programmes binaires. Nous survolons aussi plusieurs techniques d'ana-

lyse de programmes, tant statiques que dynamiques, sur lesquelles nous construirons au cours de ce document.

Le chapitre 3 détaille différentes approches d'exécution symboliques de programmes, nous y examinons les avantages et inconvénients de chacune en plus d'en expliquer le fonctionnement.

Dans le chapitre 4, nous parcourons différentes stratégies de génération automatique de cas de test qui permettront de comparer les comportements entre une solution fonctionnelle et une version soumise par un étudiant.

Au chapitre 5, nous montrons à haut-niveau l'approche de la solution proposée. Nous décrivons l'analyse du solutionnaire et des différentes soumissions étudiantes.

Le chapitre 6 contient les détails techniques d'implémentation tant de la solution proposée que des moteurs et techniques qui sont utilisés par l'outil. Nous expliquons donc comment nous avons pu étendre ANGR, un moteur d'analyse binaire existant, précisément quelles techniques et analyses du moteur ont été utilisées ainsi que leur fonctionnement.

Le chapitre 7 présente une évaluation de la performance de l'outil. Nous y mesurons l'efficacité de la solution en termes de couverture de code au travers d'instrumentation d'un corpus de programmes d'étudiants.

Nous concluons le mémoire en résumant l'efficacité des différentes analyses et approches ainsi qu'en synthétisant l'efficacité de la solution proposée. Nous proposons ensuite des pistes de travail futures.

CHAPITRE I

PEP/8 : UN SIMULATEUR DE PROCESSEUR

1.1 Une machine virtuelle pédagogique

PEP/8 est une machine virtuelle développée par J. S. Warford, professeur à l'université de Pepperdine aux États-Unis d'Amérique. Elle sert à présenter des concepts d'architectures des micro-ordinateurs (Warford et Okelberry, 2007). La machine virtuelle présente un simulateur de processeur, un jeu d'instructions et l'assembleur associé. Les utilisateurs peuvent observer leurs programmes sous la forme du code assembleur ou sous la forme du code objet assemblé (binaire). L'exécution des programmes binaires peut être tracée, déboguée et observée au travers d'une interface présentant la disposition de la mémoire et le contenu des registres à chaque pas de l'exécution.

1.2 Description technique de l'architecture, de l'assembleur et du jeu d'instruction

Le simulateur de processeur PEP/8 est une machine de von Neumann avec une architecture de 16 bits.

Le processeur dispose de 5 registres d'un mot, dont deux directement adressables. Il possède un accumulateur (`A`) et un registre d'index (`X`) dont l'utilisation est co-

dée directement dans les instructions les utilisant. Il possède également un registre de pile (*Stack Pointer* ou `SP`) avec les instructions de décalage de pile conventionnelles : `ADDSP` et `SUBSP` qui l'incrémente et le décrémente respectivement. On y retrouve aussi un registre pointeur de programme ou compteur ordinal (*Program Counter* ou `PC`) qui contient l'adresse de la prochaine instruction à exécuter. De plus, un registre d'instruction (*Instruction Register* en anglais) qui est directement inaccessible, mais qui permet à l'étudiant d'observer l'instruction décodé lors de son exécution. Un dernier registre de 4 bits existe afin de tenir les valeurs des bits d'état `N Z V C`, qui représentent respectivement si le résultat de la dernière opération était négatif (*Negative*), zéro (*Zero*), a débordé (*Overflow*) ou a causé une retenue (*Carry*).

Le jeu d'instruction comporte 39 instructions codées sur 8 bits. Les instructions qui nécessitent un opérande sont suivies directement de l'opérande sur deux octets. PEP/8 supporte 8 modes d'adressage fortement inspirés des différentes utilisations de la mémoire de langage à plus haut niveau. Le mode d'adressage d'une instruction est codé avec l'instruction lors de l'assemblage. Des directives à l'assembleur sont également disponibles afin de réserver des cases mémoires ou d'en initialiser avec des valeurs.

Le chargement d'un programme en mémoire est très simple. Le code objet (binaire) du programme est chargé directement à l'adresse 0 de la mémoire. Le simulateur ne propose pas de système de pagination mémoire ni de séparation de zones mémoire en segments ou en sections. Les programmes PEP/8 n'utilisent pas de bibliothèque partagée puisque l'assembleur ne les supporte pas et qu'aucun éditeur de lien n'est disponible. Le noyau du système d'exploitation est préalablement chargé en mémoire morte (*ROM*). Le noyau est entièrement écrit en PEP/8 et il est possible de le redéfinir. En pratique, le noyau officiel est utilisé, sauf pour les usages avancés. La redéfinition du noyau dépasse le cadre du présent mémoire.

Le jeu d'instruction de *PEP/8* est intéressant au niveau pédagogique, mais aussi au niveau du prototypage d'outils d'analyse de programmes binaires. En effet, la possibilité d'observer le code assemblé pour chaque programme permet de faire abstraction de la disponibilité du code source. Il est donc possible de construire des outils d'analyse de code binaire qui travailleront directement avec le code objet des programmes. De plus, le jeu d'instruction proposé par le simulateur PEP/8 est réaliste, bien que réduit. Il est également simple, ce qui contribue à la réalisation de tels outils en limitant l'envergure du développement pour supporter le jeu d'instruction complet d'un vrai processeur et permet donc d'accorder plus d'effort aux théories d'analyse et à leur implémentation plutôt qu'au support d'une architecture de processeur riche et complexe.

1.3 Utilisation à l'UQAM

Le cours INF2170, *Organisation des ordinateurs et assembleur*, offert dans le cadre du baccalauréat en informatique et génie logiciel de l'Université du Québec à Montréal couvre les concepts de fonctionnement d'un ordinateur à partir de différentes couches de la machine. Les étudiants sont introduits au langage d'assemblage, à l'assembleur, au code machine et aux circuits logiques. Le tout se fait avec l'aide du simulateur de processeur *PEP/8*. Les étudiants travaillent donc, tout au long de la session, avec des programmes en assembleur *PEP/8* autant sous leur forme textuelle (langage d'assemblage) que sous leur forme assemblée (binaire).

1.3.1 Corpus de travaux pratiques

Lors de ce cours, les étudiants doivent accomplir deux ou trois travaux pratiques de degrés de difficultés variés et couvrant des concepts techniques différents. L'énoncé d'un travail pratique est le même pour chaque étudiant, ils doivent donc tous

fournir un programme assembleur ayant exactement le même comportement : les résultats d'exécution sont validés par des tests automatiques déterminés par le professeur.

L'objectif de la présente recherche étant de tenter d'appliquer des techniques d'analyse de programmes binaires dans un contexte de correction de travaux, un corpus de travaux étudiant est nécessaire. Celui que nous avons utilisé est une collection de travaux pratiques remis lors de sessions d'enseignement passées du cours INF2170 à l'Université du Québec à Montréal. Pour chacun de ces groupes de travaux, nous disposons du programme solution fourni par l'enseignant ainsi que de tous les programmes remis par les étudiants. Les remises correspondent à plusieurs énoncés différents pour lesquels nous avons à notre disposition le solutionnaire de l'enseignant et chaque remise d'étudiant.

1.3.2 Correction et évaluation des travaux pratiques

Les travaux pratiques remis par les étudiants sont évalués sur plusieurs critères incluant, mais ne se limitant pas, à la fonctionnalité, la simplicité de la solution et le style. On entend par fonctionnalité le respect de la spécification donnée dans l'énoncé du travail. Les deux autres critères sont des critères plus subjectifs pour lesquels les correcteurs ont un pouvoir discrétionnaire quant à leur application. Nous nous intéresserons donc seulement à l'aspect fonctionnalité de la correction.

Dans l'état actuel des choses, les enseignants écrivent manuellement une suite de tests qui tente d'évaluer le mieux possible les fonctionnalités du programme. Ils tentent de construire un test pour évaluer chaque cas distinct de traitement du programme, en d'autres mots, chaque cas spécifié dans la spécification. Des tests doivent également être écrits pour valider la bonne gestion des cas limites et des cas d'erreurs. Ces tests sont ensuite exécutés automatiquement sur les travaux des

étudiants suivant la remise et les résultats (succès ou échec) de chaque test sont rapportés aux correcteurs.

Lors de l'inspection manuelle du code source du travail, si les correcteurs identifient des erreurs qui n'ont pas été identifiées par les tests, ils doivent eux-mêmes trouver une entrée au programme qui démontrera le comportement erroné. Non seulement c'est un procédé qui est coûteux en temps, mais il dépend de l'aptitude du correcteur, à ce moment précis, à trouver a) la logique erronée et b) comment la déclencher.

CHAPITRE II

ANALYSE DE PROGRAMMES

2.1 Introduction à l'analyse de programmes

L'analyse de programme est le processus par lequel il est possible d'analyser ou d'étudier le comportement d'un programme. Le comportement analysé peut être relatif à certaines propriétés particulières comme la correction ou la sécurité. Les objectifs de l'analyse de programmes sont généralement l'optimisation de programme ou la correction. L'optimisation de programmes cherche à améliorer la performance d'un programme, en termes d'utilisation de ressources. L'étude de la correction de programmes observe plutôt si un programme se comporte correctement par rapport à une spécification donnée. Dans le contexte de ce travail, nous nous concentrerons sur l'aspect correction.

Les analyses de programmes se classifient habituellement comme des analyses statiques ou des analyses dynamiques (ou une combinaison des deux approches). Une stratégie d'analyse est dite statique si elle raisonne sur le comportement d'un programme sans l'exécuter. Par opposition, les analyses dites dynamiques raisonnent sur un programme en observant son exécution. Une différence importante entre les deux types d'analyses est que les analyses statiques tentent d'extraire de l'information en observant un programme et que les analyses dynamiques le font plutôt en observant les traces d'exécution d'un programme.

2.1.1 Difficultés et approximation

L'analyse de programmes permet de raisonner sur les comportements d'un programme. Par contre, les questions auxquelles nous cherchons à répondre grâce à l'analyse de programmes portent sur des propriétés sémantiques non triviales. Une propriété sémantique est une propriété du comportement du programme plutôt que de sa syntaxe. Une propriété est non triviale si elle est ni vraie pour toutes fonctions calculables, ni fausse pour toutes fonctions calculables. *Le programme contient 12 fonctions* est donc une propriété syntaxique triviale. La propriété *Le programme ne retourne jamais de valeurs nulles* est sémantique. D'après (Rice, 1953), toute propriété sémantique non triviale d'un programme est indécidable.

Les algorithmes qui tentent de répondre à des questions indécidables, sans approximer, ne retournent pas toujours une réponse correcte : soit ils retournent une réponse correcte, soit ils ne terminent pas. Ce type d'algorithme peut parfois offrir de fortes assertions sur l'absence de propriétés (état du programme qui fait manifester un bogue). Dans d'autres cas, on souhaite toujours obtenir une réponse, malgré le risque qu'elle soit parfois incorrecte. Pour ce faire, les algorithmes font des approximations (des compromis) pour simplifier le problème et rendre la question décidable. Ces approximations induisent la possibilité de faux positifs ou de faux négatifs.

Le choix des approximations est fait en fonction des compromis que l'on accepte de faire sur les résultats d'analyses. Les analyses font habituellement des concessions sur le plan de la reproductibilité et sur le spectre de l'*insight* (Shoshitaishvili *et al.*, 2016).

```
div(x, y) {
    return x / y;
}

main() {
    int x = read_int();
    int y = read_int();
    int res;
    if (y != 0) {
        res = div(x, y);
        print(res);
    }
}
```

Listing 2.1: Un programme exemple.

```
main() {
    int x = read_int();
    int y = read_int();
    int res = x / y;
    print(res);
}
```

Listing 2.2: Un second programme exemple.

Reproductibilité. ou *rejouabilité*, caractéristique des analyses qui mesure la capacité à reproduire le résultat de l'analyse dans une exécution normale (concrète) du programme. Cette caractéristique varie principalement avec l'étendue de l'analyse. Les analyses qui touchent l'entièreté d'un programme ont une meilleure capacité à raisonner précisément sur *comment* déclencher une situation en particulier. Par opposition, une analyse qui se concentrerait sur seulement une partie d'un logiciel aura récolté moins d'information (ou de l'information moins précise) sur la

manière de déclencher ledit comportement. Des analyses plus locales ou d'envergures restreintes sont moins coûteuses à effectuer et permettent donc d'obtenir des résultats plus facilement au détriment de la reproductibilité des résultats. Face aux problèmes d'extensibilité de ces analyses, une analyse qui compromet la reproductibilité de ses résultats sera plus susceptible de générer des résultats erronés.

À titre d'exemple, considérons une analyse du programme au Listing 2.1 qui analyse chaque fonction séparément. Après l'analyse de la fonction `div`, elle retourne que le programme peut planter avec une division par zéro si y vaut zéro. Or, ceci est faux dans le contexte global du programme. L'analyse a donc su identifier précisément la cause de la division par zéro potentielle, mais les résultats de l'analyse n'étaient pas rejouables sur le programme à cause de la vérification sur y dans la fonction `main`.

Insight. (Köhler et Guillaume, 1928), intelligence sémantique, ou capacité à raisonner sur les causes et les effets dans un contexte donné. Par exemple, un outil de tests à données aléatoires (*Fuzzing*) peut déterminer qu'une certaine entrée fait planter un programme, mais sera incapable de dire précisément quelles parties (voire bits) de l'entrée en sont responsables. Une analyse qui a suffisamment d'*insight* en sera capable.

Par exemple, pour une analyse sur le Listing 2.2 qui manque d'*insight* peut dire que l'entrée `0 0` (pour x et y respectivement) cause une division par zéro, mais sera incapable de déterminer que la variable qui cause la division par zéro est la variable y.

Les analyses à forte intelligence sémantique doivent maintenir une grande quantité d'information sur les programmes analysés. Lorsque les programmes à analyser se

complexifient, il devient impossible pour les analyses de maintenir l'entièreté de cette information et elles doivent donc réduire la quantité d'information qu'elles enregistrent, ce qui entraîne une réduction de l'*insight*. Un exemple d'une telle situation serait une analyse de valeur qui sauvegarderait des intervalles de valeurs possibles plutôt que la liste exhaustive des valeurs exactes que peut prendre chaque variable (Shoshitaishvili *et al.*, 2016).

2.1.2 Le problème de l'environnement

Historiquement, la gestion de l'environnement d'exécution est un problème récurrent pour les outils d'analyse de programmes. On entend par environnement d'exécution tout le contexte dans lequel le programme s'exécute : Ligne de commande, système d'exploitation, bibliothèques (Cha *et al.*, 2012), système de fichiers, communication réseau (Cadar *et al.*, 2008), etc. Effectivement, la tâche de chargement d'un logiciel en mémoire, ainsi que son interaction avec son environnement, est complexe et nécessite l'interaction de plusieurs composantes d'un système. Afin d'être en mesure d'analyser un logiciel qui interagit (ou utilise) son environnement, il est nécessaire de modéliser cet environnement et ces interactions. La complexité de ces comportements rend la tâche de modélisation complète aussi (ou plus) complexe que l'écriture même de tous ces aspects. De plus, ce sont des parties de l'exécution d'un programme qui sont difficiles à prévoir et à reproduire. Par exemple, il est difficile de prévoir une erreur d'allocation de mémoire par le noyau ou encore la valeur de retour d'un appel à un générateur de nombres aléatoires.

Évidemment, il n'est pas réaliste de complètement modéliser précisément un système d'exploitation. L'approche habituelle est de modéliser de façon simplifiée un sous-ensemble choisi d'appels système et de bibliothèques. C'est dans ces appels modélisés que seront gérées les interactions avec l'environnement. Principalement,

les auteurs d'outils penchent pour une modélisation simple qui permet d'analyser les cas les plus habituels d'utilisation (Cadar *et al.*, 2008) et les comportements d'échecs fréquents (Cha *et al.*, 2012). Par exemple, dans MAYHEM, l'ouverture d'un fichier est modélisée par une couche de virtualisation qui retournera des résultats basés sur une trace d'exécution. Le contenu du fichier, quant à lui, est représenté comme un espace mémoire isolé des autres. Nous verrons au chapitre 6 comment ce problème est atténué dans le cas du simulateur PEP/8.

2.1.3 Analyse statique

Domaine abstrait

Le fait de raisonner sur le comportement d'un programme sans avoir besoin de l'exécuter a certains avantages. Principalement, ces avantages apparaissent du fait que les instructions du programme ne sont pas exécutées, mais seulement étudiées. Ceci permet d'utiliser un domaine de valeurs qui est différent de celui de l'exécution concrète (les 1 et les 0 en mémoire). On parle alors d'un *domaine abstrait.* Il est possible de simplifier un domaine de valeurs afin de réduire la complexité de l'analyse. Un exemple serait de maintenir seulement le signe des valeurs (positif, négatif ou zéro) pour chaque variable plutôt que leur valeur réelle. Il est aussi possible d'utiliser l'ensemble des valeurs possibles des variables au court de l'analyse, qui permettrait de raisonner à la fois sur toutes les exécutions possibles du programme.

Évidemment, la réduction d'un domaine (comme l'exemple des signes des valeurs ci-haut) entraîne une perte de précision dans l'analyse. Ce domaine peut être suffisant pour répondre à certaines questions sur certains programmes, mais insuffisant pour d'autres. L'exemple des signes des valeurs est suffisant pour connaître le signe des valeurs pour tout point d'un programme qui n'utilise que des multipli-

cations, mais devient insuffisant dès que des additions sont utilisées. Les résultats de l'analyse peuvent donc être faux si des opérations d'additions sont utilisées. La réduction de la précision du domaine, afin de réduire la complexité de l'analyse, entraine une éventuelle perte d'*insight* ou de reproductibilité, ou les deux. Une augmentation du domaine induit par contre des analyses plus complexes, plus couteuses.

Certaines analyses statiques, qui raisonnent globalement, ne rendent pas de résultats si l'analyse n'est pas complétée. Les analyses qui explorent exhaustivement l'espace d'état du programme, ou les analyses qui représentent les programmes comme des formules logiques en sont des exemples. Ces formes d'analyses font face à un problème additionnel : les procédures difficiles à modéliser (Cha *et al.*, 2012). Pour analyser le comportement complet d'un appel système, il faut avoir un modèle du système d'exploitation, ce qui n'est pas réaliste en pratique.

Types d'analyses statiques

Afin d'illustrer le spectre des différentes techniques d'analyses statiques, nous présentons ici un aperçu de certains types d'analyses statiques sur lesquelles sera construite la suite du document.

Analyse de flot de contrôle. Analyse qui observe les transferts de flots d'un programme pour générer un graphe de flot de contrôle (GFC, *Control-Flow Graph* ou *CFG* en anglais). Il s'agit d'un graphe orienté qui présente les différents chemins d'exécution possibles dans un programme. Les noeuds représentent les blocs de base d'instructions (séquences linéaires d'instructions sans branchements). Les arrêtes représentent des transferts de flot d'exécution possibles (branchements potentiels). La Figure 2.1 présente un exemple de GFC. Les noeuds a, b, c et d sont

des blocs de base d'instructions. Les flèches noires représentent les branchements inconditionnels, les vertes et les rouges représentent respectivement les branches *vrai* et *faux* d'un branchement conditionnel.

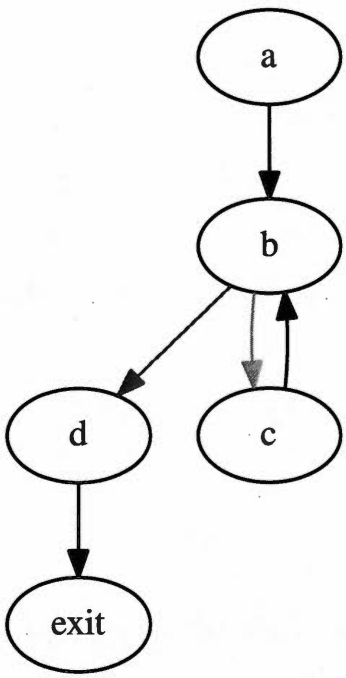


Figure 2.1: Exemple de graphe de flot de contrôle.

Analyse de flot de données. Tire profit du GFC afin de trouver l'ensemble des valeurs possibles des variables pour tout point d'un programme en observant la propagation des valeurs.

Interprétation abstraite. Approximation correcte d'une partie d'un programme en faisant une exécution partielle du programme sur un domaine abstrait afin d'en extraire de la sémantique.

Vérification de modèle. Technique de vérification formelle qui vérifie que le modèle d'un programme est conforme à une spécification donnée. Les états possibles du programme sont représentés sous la forme d'une formule logique. La spécification est, elle aussi, transformée en formule logique. Il est alors possible de vérifier que la formule logique du programme satisfait la formule logique de la spécification donnée.

2.1.4 Analyse dynamique

Les analyses dynamiques de programmes exploitent l'information des traces d'exécution du programme. L'observation du comportement du programme peut se faire a posteriori, comme dans le cas des tests (définis en section 4.1), ou pendant l'exécution. Il est également possible de modifier l'environnement dans lequel évolue le programme ou d'enrichir le programme d'instructions additionnelles afin d'en extraire de l'information. Par contre, l'analyse dynamique, dépendant de l'exécution du programme, doit typiquement raisonner sur un seul chemin d'exécution à la fois. Les analyses dynamiques offrent un haut niveau de rejouabilité grâce à l'information additionnelle que la trace d'exécution apporte.

2.1.5 Analyse de programmes binaires

Il y a plusieurs raisons d'analyser le code binaire d'un programme plutôt que son code source. Une première est tout simplement l'absence du code source. Certains distribuent des logiciels sans fournir le code source associé : logiciels

à source fermés, logiciels malveillants, etc. D'autres ont été directement écrits en assembleur : code légataire, optimisations manuelles, assembleur embarqué dans un programme en C, code impénétrable (Balakrishnan *et al.*, 2008). Une autre raison à l'analyse du code binaire est que les détails de bas niveau ont une importance dans certains cas. Par exemple, dans la recherche de vulnérabilités, de l'exploitation automatique de vulnérabilités (Song *et al.*, 2008; Cha *et al.*, 2012; Shoshitaishvili *et al.*, 2016).

Analyser directement le code binaire d'un programme introduit une difficulté additionnelle : l'absence de sémantique de haut niveau.

L'organisation du programme. Les constructions d'organisation du code, comme les fonctions, ne sont pas clairement identifiables dans le code binaire. Il n'y a pas non plus de moyen d'être sûr que les octets qui correspondent à l'instruction `CALL` sont véritablement une instruction ou s'il s'agit simplement d'octets de données (ou les deux). Il y a également une multitude d'instructions (ou de séquences d'instructions) qui peuvent se comporter comme une instruction `CALL` (`PUSH; BR` par exemple). L'organisation du code est donc beaucoup plus complexe à interpréter au niveau binaire.

Tampons et variables. Les variables, les tampons, les tableaux et les autres structures d'agrégation n'existent pas au niveau binaire. La seule abstraction structurelle des données est la mémoire.

Typage. En format binaire, toutes les informations de typage de haut niveau ont été effacées. Il est donc impossible de retrouver ce qu'était le type d'une variable dans le code source initial. Il devient donc difficile de différencier une valeur directe d'un pointeur sur de la donnée, d'un pointeur de fonction.

Pour la suite du document, nous discuterons d'analyse de programme dans le contexte de programmes binaires seulement.

2.2 Analyses statiques de programmes binaires

2.2.1 Reconstruction de graphes de flot de contrôle

Toutes analyses statiques qui voudraient modéliser les dépendances ou les transferts de données globalement dans un programme nécessiteront initialement de connaître les transferts de flots de contrôles du programme (Allen, 1970). La première analyse effectuée sur un programme est donc souvent une reconstruction du graphe de flots de contrôle du programme. Ceci permettra ensuite d'effectuer d'autres analyses de dépendance de données.

La reconstruction de base d'un GFC s'implémente avec un algorithme récursif plutôt simple. En commençant par le bloc de la première instruction, on désassemble le bloc d'instructions, on l'analyse pour identifier les noeuds successeurs potentiels (sorties possibles du bloc, destinations possibles des branchements) puis ajoute ces nouveaux blocs à une liste de travail. En prenant récursivement chaque bloc de la liste de travail et en répétant le travail tant qu'il reste des blocs à traiter, on peut reconstruire un GFC de tous les blocs atteignables par branchements directs.

Le défi de la reconstruction des graphes de flots de contrôle se cache dans les branchements indirects. On entend par branchement indirect un branchement dont la destination n'est pas codée dans l'instruction. La destination d'un branchement indirect peut dépendre de plusieurs facteurs. La destination peut être calculée comme dans le cas d'une instruction `switch`, sensible au contexte comme dans le cas d'une fonction de rappel ou dépendre d'un objet comme dans le cas d'un appel polymorphe. Différentes analyses et techniques se spécialisent dans la résolution

de différents types de branchements indirects (Cifuentes et Van Emmerik, 1999; Schwarz *et al.*, 2002; Kruegel *et al.*, 2004; Kinder et Veith, 2008).

La technique de reconstruction du graphe de flot de contrôle a deux propriétés qui permettent de qualifier son efficacité [1] (Shoshitaishvili *et al.*, 2016) :

La correction L'analyse de reconstruction est correcte si tous les transferts de flot du programme sont représentés dans le graphe généré. Lorsque cette analyse résout les destinations possibles d'un branchement comme un sous-ensemble de ses possibilités, la correction du graphe diminue.

La complétude Par opposition, une analyse est complète si tous les transferts de flots représentés sont réellement possibles dans l'exécution du programme.

C'est-à-dire, pour un GFC original $G = (V, E)$ et un graphe reconstruit $G' = (V', E')$, G' est correct *ssi* $E \subseteq E'$ et $V \subseteq V'$ et est complet *ssi* $E' \subseteq E$ et $V' \subseteq V$.

Un algorithme qui n'est pas correct manquera des destinations possibles pour des branchements du programme. Les destinations manquées ne seront pas analysées pour trouver leurs successeurs et ainsi de suite. Celà produit un GFC dans lequel une partie du code et des transferts de contrôle du programme ne sont pas représentés. La correction peut être interprétée comme le ratio de vrais positifs d'une analyse de reconstruction de GFC. Un algorithme qui tend vers l'incomplétude représentera des branchements impossibles. La destination de ces branchements

1. Xu et Rothermel (2009) utilisent la notation inverse où un graphe est complet lorsque tous les transferts potentiels du programme sont représentés et correct lorsque tous les transferts représentés sont possibles. Comme le travail présenté utilise ANGR comme suite d'analyse, nous utiliserons la définition présentée par Shoshitaishvili *et al.* (2016).

sera faussement analysée afin d'en extraire les successeurs. Ce comportement induit un GFC erroné. Les manquements en complétude s'apparentent à la notion de faux positif (Shoshitaishvili *et al.*, 2016).

2.2.2 Analyse d'ensembles de valeurs

Initialement présentée par Balakrishnan et Reps (2004) comme une analyse sur les patrons d'accès à la mémoire afin de construire des tables de symboles, l'analyse d'ensemble de valeurs (ou *Value-Set Analysis*, *VSA*, en anglais) est utile pour la suite des analyses sur les programmes binaires. Principalement, cette analyse de résolution de valeurs sera utile dans la résolution des branchements indirects.

L'analyse d'ensemble de valeurs pour un programme binaire a pour objectif d'obtenir une surapproximation serrée des valeurs possibles de toutes les variables pour tous les points du programme. En pratique, VSA est une interprétation abstraite du programme qui maintient les ensembles de valeurs possibles des pointeurs et des variables simultanément. Dans le cas d'un programme binaire, analyser les valeurs et les pointeurs à la fois est important puisqu'ils sont indistinguables sans leur contexte d'utilisation. VSA parvient à faire la distinction entre les deux en observant la façon dont chaque variable est utilisée. Un déréférencement, par exemple, indique un pointeur. En unissant les informations d'utilisation aux ensembles de valeurs possibles des variables, il est possible de resserrer davantage la surapproximation faite.

L'information recueillie par l'analyse d'ensemble de valeurs permet non seulement de trouver l'emplacement des variables dans le programme, mais également de générer une surapproximation serrée des valeurs possibles de ces variables pour tout point du programme. Ceci s'avère utile lors de la résolution des branchements indirects (Shoshitaishvili *et al.*, 2016). Notamment, réduire la plage possible des

variables réduit indirectement la charge sur le solveur associé à l'exécution symbolique (qui sera discuté aux points suivants) (Cha *et al.*, 2012). De plus, l'approximation serrée des valeurs utilisées comme branchements indirects augmente la complétude du GFC (Shoshitaishvili *et al.*, 2016).

2.3 Analyses dynamiques de programmes binaires

2.3.1 Exécution concrète

On appelle exécution concrète le fait d'exécuter un programme dans un environnement le plus normal possible, mais tout en récoltant des informations sur son exécution. Le programme est donc habituellement exécuté sur son domaine normal et analysé un chemin à la fois. Il faut donc typiquement fournir une entrée fonctionnelle (un cas d'utilisation normal) ou un cas de test. L'exécution concrète peut varier du comptage d'instruction à la génération de traces (plus ou moins ciblées) au *fuzzing*.

Instrumentation

L'instrumentation de programme est une technique qui permet de surveiller l'exécution d'un programme. Certaines approches insèrent des instructions additionnelles dans un programme qui ajoutent de la logique de supervision ou de traçage dans le programme ciblé, d'autres utilisent une approche similaire aux débogueurs afin de surveiller le programme analysé. L'instrumentation est particulièrement utile pour générer des traces ciblées, par exemple la liste de toutes les adresses des instructions exécutées au cours de l'exécution d'un chemin dans un programme.

CHAPITRE III

EXÉCUTION SYMBOLIQUE

3.1 Exécution symbolique statique

L'exécution symbolique statique (ESS) est une technique de vérification formelle dans laquelle on représente l'exécution d'un programme comme une formule logique. Cette formule représente les propriétés du programme pour les différents chemins d'exécution. Cette technique est principalement utilisée dans le cadre de la recherche de vulnérabilités (ou de bogues) (Avgerinos *et al.*, 2014b). En fait, la nature statique et globale de cette analyse (l'ESS vérifie un programme et non un chemin dans un programme) la rend plus utile pour prouver l'absence d'une vulnérabilité que pour la détecter (Avgerinos *et al.*, 2014b; Shoshitaishvili *et al.*, 2016). Les états recherchés, comme une vulnérabilité ou un bogue en particulier, sont codés sous la forme d'une assertion logique qui rendront fausse la formule logique du programme si la sécurité du programme est violée. La formule est ensuite vérifiée par un solveur.

Avantages. L'ESS code toutes les possibilités d'exécution dans la même formule logique. Cette approche porte plusieurs avantages considérables par rapport à d'autres techniques d'analyse symboliques. Notamment, le fait de pouvoir coder et résumer efficacement l'effet d'une boucle ou des différents chemins d'un

branchement conditionnel au point de confluence (point du programme où les différentes branches d'exécution se rejoignent) (Avgerinos *et al.*, 2014b) via une formule compacte offre des gains de performance considérables (Flanagan et Saxe, 2001; Leino, 2005; Barnett et Leino, 2005; Jager et Brumley, 2010). Il faut par contre considérer que l'exploration des différents chemins doit tout de même être gérée par le solveur.

```
y = 25;
if (x < 18) {
    y = 5;
} else if (x > 60) {
    y = 10;
}
```

Listing 3.1: Segment de programme exemple pour l'exécution symbolique statique.

Afin d'illustrer ces avantages, la Figure 3.1 présente l'évolution du magasin de variables symboliques lors de l'exécution symbolique statique du programme en Listing 3.1. Il est particulièrement intéressant de remarquer comment l'ESS résume l'état des deux côtés de chaque branchement dans le programme d'exemple. Les branchements sont codés sous la forme *If-Then-Else*, comme présentée au noeud B5. Aussi, la valeur symbolique de y au noeud B6 présente la capacité de l'ESS à joindre différentes branches d'exécutions possibles en simplifiant les conditions codées sous la forme *If-Then-Else*.

Limitations. Malgré ses avantages considérables face à d'autres techniques d'analyse, l'ESS souffre de problème d'extensibilité lorsqu'utilisée pour l'analyse de programmes plus gros et plus complexes (Beyer *et al.*, 2007; Kuznetsov *et al.*, 2012). À la racine de cette limitation est le problème de la *complétion*. Puisque l'ESS est un algorithme qui raisonne sur le programme et non sur des chemins dans

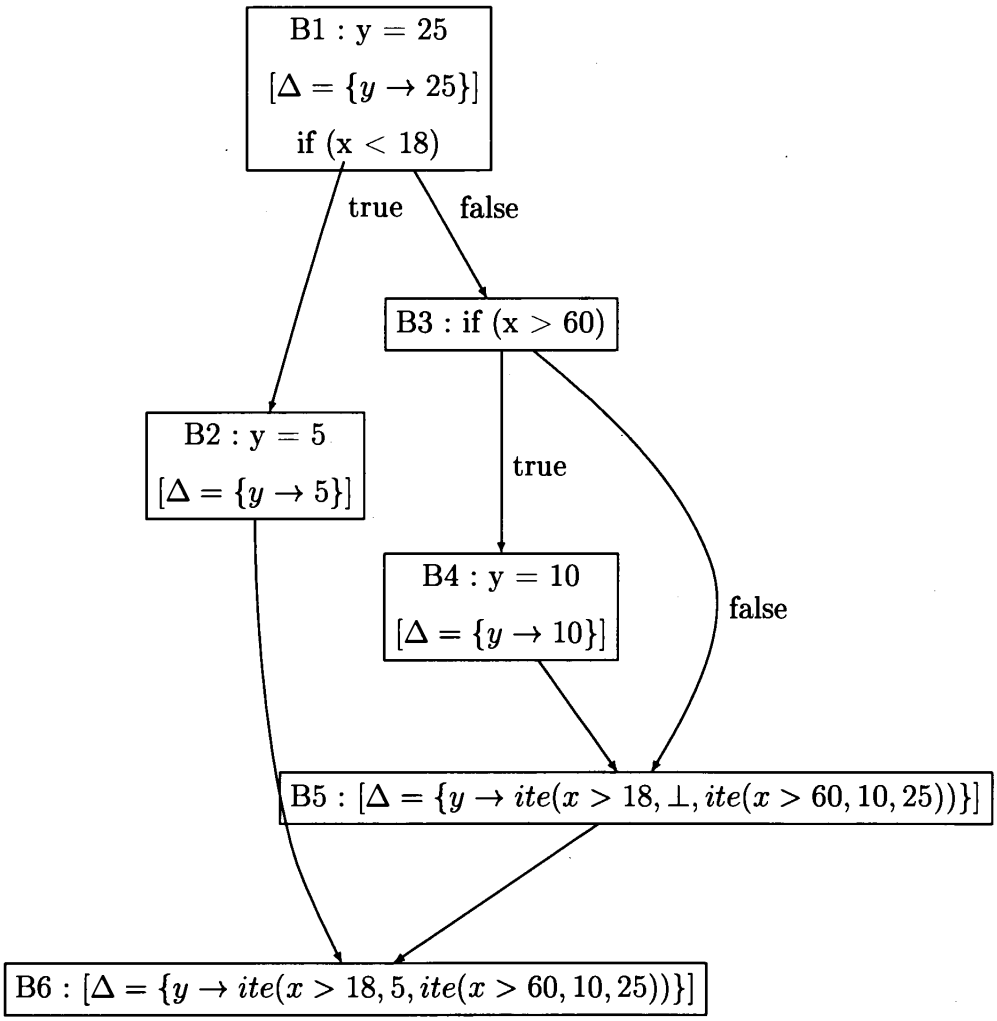


Figure 3.1: Magasin symbolique lors de l'exécution symbolique statique du segment de programme en Listing 3.1.

un programme, l'analyse doit se terminer sur le programme en entier avant de pouvoir rendre des résultats. Or, certains obstacles nuisent à l'analyse complète d'applications complexes. Certains aspects de l'exécution d'un programme sont difficiles à modéliser particulièrement les appels système, les bibliothèques et l'environnement d'exécution (comme discuté en sous-section 2.1.2). De plus, la gestion des programmes cycliques ou récursifs est difficile au niveau de l'ESS. Combien

de fois doit-on dérouler chaque boucle (Avgerinos *et al.*, 2014b) ? Comment gérer efficacement la récursion dans les formules logiques (Dillig *et al.*, 2008) ?

3.2 Exécution symbolique dynamique

L'exécution symbolique dynamique (ESD), par opposition à l'approche statique, s'effectue lors de l'interprétation du programme, une branche d'exécution à la fois. Cette analyse explore le programme et génère les prédicats de chemins (formules logiques qui sont vraies que pour un chemin d'exécution) pour chaque chemin qu'elle rencontre indépendamment. Les prédicats de chemins se forment en contraignant les valeurs possibles des variables abstraites au fur et à mesure que l'on interprète un chemin d'exécution (en ajoutant des conditions ou des bornes aux variables dont le domaine est contraint). Cette approche encaisse un surcoût à l'exécution en échange de gains de performances à la résolution des prédicats.

L'exécution symbolique dynamique s'applique sur un langage impératif. Elle exécute les instructions une à la fois en appliquant les effets de chaque instruction une à une. À chaque étape de l'exécution, l'exécuteur sauvegarde dans son *état* l'adresse de la prochaine instruction à exécuter, son prédicat de chemin, ainsi que les valeurs symboliques de toutes les variables. Lorsque l'algorithme rencontre un branchement conditionnel, l'exécuteur se clone. Chaque clone ajoute les conditions aux variables qui influent sur la branche prise. En d'autres mots, l'exécuteur de la branche *vrai* ajoutera à son prédicat de chemin les bornes possibles aux variables qui feront en sorte que cette branche sera empruntée lors de l'exécution de programme. L'autre exécuteur (celui de la branche *faux*) ajoutera à son prédicat de chemin les bornes complémentaires. La Figure 3.2 montre les contraintes ajoutées au prédicat de chemin lors des branchements conditionnels lors de l'ESD.

```
int i = 0;
int x = read_int();
if (x > 0) {
    i = i - 1;
} else {
    i = i + 2;
}
if (x == 42) {
    x = x + i
    i = 42;
}
```

Listing 3.2: Segment de programme exemple pour l'exécution symbolique dynamique.

Avantages. L'exécution symbolique dynamique est simple à implémenter par rapport à son homologue statique. La résolution des prédicats de chemins qu'elle accumule est rapide comparativement à l'ESS. La nature dynamique de cette analyse offre une grande reproductibilité à ses résultats (Bounimova *et al.*, 2013). Aussi, le problème de la complétion de l'ESS est inexistant pour un moteur d'ESD puisqu'elle agit et évalue localement (un chemin d'exécution à la fois). Il est également facile pour un moteur d'ESD de contourner le problème des aspects difficiles à modéliser (Avgerinos *et al.*, 2014b). Par exemple, un appel système peut être exécuté concrètement, simulé, substitué ou simplement ignoré sans nuire aux autres branches de l'exécution. De plus, il est possible d'appliquer des heuristiques de priorisation de chemins à exécuter afin de cibler certains états recherchés ou d'en éviter.

Limitations. La plus grande limitation de l'exécution symbolique dynamique est sans aucun doute l'explosion combinatoire du nombre d'états (ou de che-

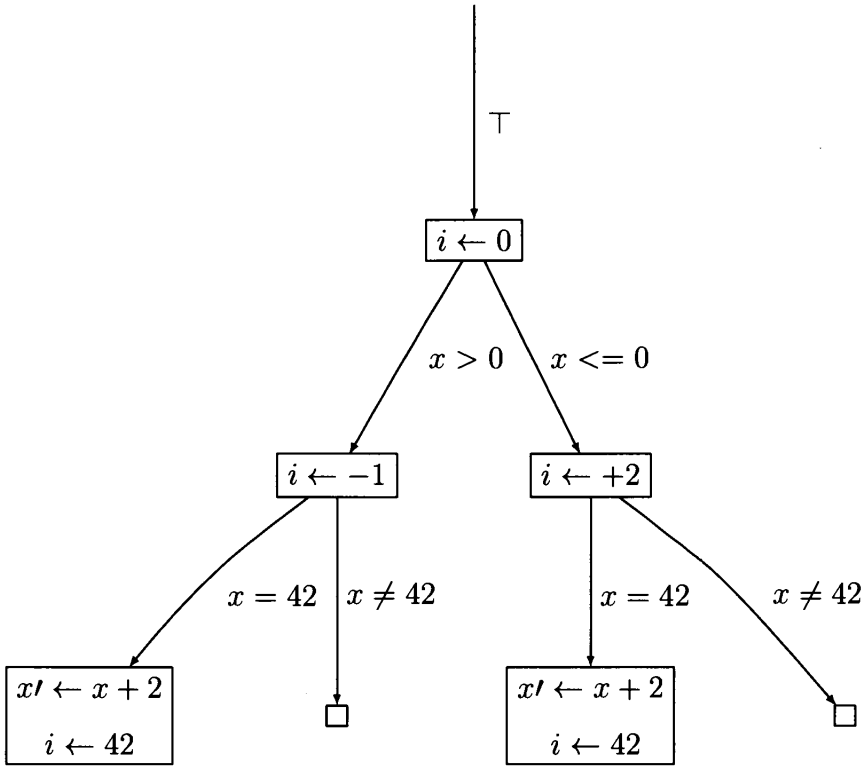


Figure 3.2: Visualisation des contraintes ajoutées aux branchements conditionnels lors de l'exécution symbolique dynamique du segment de programme en Listing 3.2.

mins) à analyser (Cadar *et al.*, 2008; Cha *et al.*, 2012; Kuznetsov *et al.*, 2012; Avgerinos *et al.*, 2014b; Su *et al.*, 2016; Shoshitaishvili *et al.*, 2016). Pour comprendre comment cela se produit, il suffit de voir que pour chaque tour de boucle conditionnelle, un nouvel exécuteur est produit. Une boucle génère donc autant de nouveaux exécuteurs que son nombre de tours maximal, et ce sans compter les potentiels branchements conditionnels dans cette boucle. À titre illustratif, la Figure 3.2 montre qu'il faut quatre exécuteurs pour gérer seulement deux branchements conditionnels. Le nombre d'exécuteurs requis augmente exponentiellement. Le nombre d'exécuteurs et d'états mémoires à maintenir lors de l'exécution sym-

bolique dynamique est assez pour ralentir déraisonnablement et même rendre impossible l'analyse de programmes complexes par ESD naïve. Certaines approches de priorisation de chemins (comme une exploration en profondeurs plutôt qu'en largeur) ou de fusion d'états (discutés en section 3.4) tentent donc de limiter les impacts de l'explosion combinatoire du nombre d'exécuteurs.

3.3 Exécution concolique

L'exécution concolique (*concrète* et *symbolique*), ou exécution *hors-ligne*, se veut une superposition de l'exécution concrète et de l'exécution symbolique dynamique. En démarrant d'une trace d'exécution concrète, on exécute symboliquement le programme en suivant cette trace. L'exécution symbolique permet de généraliser la trace concrète en un prédicat qui caractérise son chemin dans le programme. L'exécution d'un seul chemin du programme est effectuée symboliquement à la fois. L'exécution concolique est avantageuse puisqu'elle n'a pas à maintenir tous les états à éventuellement exécuter comme dans le cas de l'ESD. L'inconvénient principal des approches d'exécutions concoliques est la perte de performance considérable due à la réexécution d'états. Pour analyser les deux chemins issus d'un branchement conditionnel qui se trouverait profondément dans la trace d'un programme (quelques millions d'instructions par exemple), on devra réexécuter toutes les instructions qui mènent au branchement lors de la visite du deuxième chemin. Le problème se répète récursivement plus l'exploration atteint des points profonds dans les chemins d'exécutions du programme.

3.3.1 Exécution symbolique hybride

Dans MAYHEM (Cha *et al.*, 2012), les auteurs proposent un modèle d'exécution symbolique hybride qui allie l'exécution symbolique *en ligne* et *hors-ligne*. L'idée

est que l'exécuteur symbolique hybride agit comme gestionnaire d'exécution symbolique et fait alterner des phases d'exécution *en ligne* et *hors-ligne* de manière à ne pas surcharger la mémoire. Le moteur surveille la mémoire alors que l'exécution dynamique symbolique se produit. Lorsque l'utilisation de la mémoire atteint un certain seuil, le superviseur génère des points de sauvegardes pour certains exécuteurs et détruit ces exécuteurs et leur état. Les informations sauvegardées pour chaque exécuteur arrêté sont : l'adresse actuelle, le prédicat de chemin, le magasin symbolique ainsi que des informations de profilage. Une fois l'exploration *en ligne* terminée, MAYHEM restaure les progressions sauvegardées. Pour ce faire, l'exécuteur est recréé à partir des informations du point de sauvegarde. La restauration de l'état mémoire de l'exécuteur est faite en deux étapes. Le prédicat de chemin est d'abord résolu par le solveur SMT afin de générer une entrée au programme qui prendra le même chemin d'exécution. Le programme est ensuite exécuté *hors-ligne* (concolique) avec les entrées générées jusqu'au point de la sauvegarde. L'état mémoire de l'exécuteur est alors restauré comme avant la sauvegarde. L'exécuteur symbolique *en ligne* peut alors continuer l'exécution.

3.4 *Veritesting* et la limitation de l'explosion combinatoire

Veritesting est une déclinaison de l'exécution symbolique dynamique qui tire avantage de l'exécution symbolique statique lorsqu'il est opportun (Avgerinos *et al.*, 2014b). L'objectif est de profiter de la capacité de l'ESS de résumer l'effet des boucles et des branchements aux points de confluence de ceux-ci. Cela résulte en une meilleure couverture de chemin en échange de requêtes plus coûteuses au solveur. En pratique, *Veritesting* montre des résultats concrètement meilleurs que l'ESS ou l'ESD seules (Avgerinos *et al.*, 2014b; Shoshitaishvili *et al.*, 2016).

La stratégie consiste à interrompre l'exécuteur symbolique dynamique lorsqu'il

atteint un branchement symbolique. Plutôt que d'immédiatement cloner l'exécuteur pour couvrir les deux branches indépendamment, *Veritesting* introduit ici une passe d'exécution symbolique statique locale. On identifie une frontière difficile à modéliser (points de transitions) pour l'ESS, délimitée principalement par les appels système, les appels de bibliothèques et les retours d'appels de fonction. Ces points de transitions sont identifiés sur un GFC étendu. On nourrit l'exécuteur symbolique statique avec le prédicat de chemin actuel et on exécute l'ESS sur le GFC étendu. Par la suite, on obtient un GFC étendu qui contient les prédicats de chemins pour chaque noeud du graphe. De cette façon, il devient possible d'identifier clairement pour quels chemins des exécuteurs symboliques dynamiques doivent réellement être créés. Celà est possible parce que l'ESS est en mesure de synthétiser l'effet de tous les chemins à l'intérieur des frontières dures établies et donc dire précisément combien et quels exécuteurs sont nécessaires pour couvrir tous les noeuds à l'intérieur de ces frontières.

CHAPITRE IV

GÉNÉRATION AUTOMATIQUE DE TESTS

4.1 Cas de test et suite de tests

Afin de bien comprendre la génération de cas de test, on se doit de bien définir, ce qu'est un cas de test ainsi que les suites de tests, de même que le concept de couverture qui leur est associé.

Un cas de test est un ensemble de spécifications sur l'exécution d'un logiciel jumelé aux résultats attendus. On retrouve généralement spécifiés dans un cas de test les conditions d'exécution, les entrées données aux programmes, la procédure à exécuter ainsi que l'ensemble des résultats et sorties du programme. Une suite de tests est un ensemble de cas de test regroupés pour montrer, prouver, ou valider un ensemble de comportements.

Les cas de test (ou les suites de tests) sont utiles pour assurer qu'un programme présente un comportement spécifié. Ce genre de tests est souvent utilisé pour vérifier les nouvelles versions d'un programme et s'assurer du maintien d'un comportement correct entre les versions. Ce type d'utilisation est appelé tests de régression. Les cas de test peuvent aussi être utilisés comme aide au développement, pour spécifier le comportement attendu avant le développement de la procédure qui introduira le comportement (Beck, 2003). Celà permet de s'assurer du respect

de la spécification pendant la phase de développement. Les cas de test sont utiles dans de nombreuses phases de la vie d'un logiciel, mais dans le cas du présent travail, nous soulignons surtout l'importance des tests pour valider le respect d'une spécification par un programme.

4.2 Validation de couverture

La couverture d'un test sur un programme est la partie du programme qui influe sur le résultat du test. Ce qui s'apparente (sans représenter exactement) aux comportements du programme que le test vérifie. Historiquement, la couverture d'une suite de tests est mesurée de différentes façons selon l'objectif et le contexte des tests.

Couverture de code. Représente l'ensemble des unitées de code qui sont exécutés lors de l'exécution de la suite de tests. Certains tracent les instructions exécutées, d'autres les blocs de codes, ou alors les noeuds dans le GFC. Ultimement, ces approches tracent et mesurent quelles instructions du programme ont été exécutées.

Couverture de branchements. Ou la couverture de transitions suit l'idée de la couverture de code, mais s'intéresse aux transferts de flot d'exécution effectués plutôt qu'aux instructions exécutées. Il s'agit donc de la couverture des arêtes du GFC.

Couverture de gardes. S'intéresse aux transferts de flots, mais également aux pré- et post-conditions des différents branchements. Cette approche offre une meilleure couverture des spécifications que les autres approches (Rayadurgam et Heimdahl, 2001), mais nécessite de connaître les sémantiques de haut niveau pour

faire varier atomiquement les pré- et post-conditions des branchements. Il est donc difficile d'appliquer ce type de mesure de couverture dans le cas de la génération de tests sur un programme binaire.

4.3 Génération de suites de tests

4.3.1 Génération automatique par vérification de modèle

La génération de tests automatique par vérification de modèle (*model-checking*) tente d'utiliser un modèle du programme pour trouver les cas à tester et pour générer les tests associés (Rayadurgam et Heimdahl, 2001; Gargantini et Heitmeyer, 1999; Offutt *et al.*, 2003; Philipps *et al.*, 2003). Puisque le vérificateur est en mesure de trouver des contre-exemples possibles à une formule représentant un état du programme, il suffit de demander un contre-exemple à la non-satisfiabilité de l'état ciblé. En d'autres mots, il faut inverser les conditions d'atteinte à un état et demander un contre-exemple à l'outil de vérification. Ce contre-exemple sera alors un exemple d'entrée qui permettra l'atteinte d'un état ciblé. En répétant ce procédé pour chaque état du modèle, on obtient une suite de tests couvrant chaque état de la spécification du programme.

Deux inconvénients apparaissent en voulant appliquer cette approche à la génération automatique de tests pour un programme binaire. Premièrement, l'approche nécessite la modélisation de la spécification. Dans le cadre du travail actuel, la spécification du programme est inconnue. Il nous est donc impossible de construire un modèle basé sur cette spécification. Deuxièmement, le fait de traiter un modèle et non l'implémentation réelle du programme analysé mène à une couverture incomplète et même une suite de tests potentiellement erronée. Par exemple, le modèle de la spécification ne contient pas les détails d'implémentation du langage dans lequel elle sera implémentée (Bloem *et al.*, 2014). Certaines vérifications, comme

les tests de pointeurs nuls, n'apparaissent pas nécessairement dans la spécification du programme. Les détails d'exécution ne sont pas toujours tous présentés au niveau de la spécification. C'est plutôt au niveau de l'instruction binaire que la vérité se trouve.

4.3.2 Génération automatique par rétroaction

La génération automatique de tests par rétroaction est semblable au *fuzzing* dirigé, où les effets des entrées générées sur le programme sont utilisés pour guider les prochaines générations. La génération automatique de tests par rétroaction (Pacheco et Ernst, 2007; Robinson *et al.*, 2011; Dimjaševic, 2013) choisit aléatoirement des constructeurs et des méthodes du programme à invoquer. En utilisant la rétroaction (ou *feedback*) du programme (l'état du programme et des objets et variables), il est possible de guider les mutations à appliquer pour la génération du prochain cas de tests. La rétroaction du programme est également pertinente pour valider la couverture ajoutée des nouveaux tests mutés.

La similarité de stratégie entre cette approche et le *fuzzing* fait apparaître le même genre de limitations de couverture. Aucune intelligence sémantique ou information sur l'ensemble du programme n'est utilisée par cette approche. L'application de mutations successives sur des cas de tests ne garantit malheureusement pas une couverture totale éventuelle. De plus, cette approche nécessite l'accès au code source du programme pour connaître les constructeurs et méthodes qu'il est possible d'invoquer. Le code source l'informe aussi sur les objets à tracer et leur état. Ni les définitions des objets ni les méthodes qui s'appliquent sur ceux-ci ne nous sont disponibles.

4.3.3 Augmentation automatique de suites de tests

Une autre avenue de génération de tests automatique explorée par Bloem *et al.* (2014) et Robinson *et al.* (2011) est celle d'analyser le comportement d'une suite de tests existante pour en étendre la couverture. L'approche se résume en quelques étapes. Tout d'abord, chaque test existant est exécuté concrètement sur un programme minimalement instrumenté pour enregistrer la couverture de code initiale de chacun. Par la suite, les tests sont réexécutés afin d'enregistrer les prédicats (conditions) de chaque branchement. Ces prédicats sont alors inversés ou modifiés un à un puis résolus par un solveur afin de générer des tests explorant des nouveaux chemins dans le programme testé, augmentant donc la couverture de code de la suite de test.

L'inconvénient majeur de cette approche est la dépendance à la suite de tests initiaux qui seront mutés pour augmenter la couverture. Malheureusement les approches discutées n'utilisent pas toutes les analyses de programmes binaires discutées plus tôt pour enrichir leurs résultats.

4.3.4 Génération par exécution symbolique ou concolique

À notre connaissance, l'approche la plus appropriée de génération de tests pour le contexte ciblé est celle présentée dans KLEE (Cadar *et al.*, 2008) et étendue par des outils comme KLOVER (Li *et al.*, 2011). L'idée principale est d'utiliser l'exécution symbolique dynamique pour explorer les différents états du programme et leur prédicat de chemin. La résolution de ces prédicats de chemins fournira une entrée qui permettra de générer un cas de test pour chaque chemin.

KLEE commence par recompiler le programme avec le compilateur LLVM pour C GNU afin de générer de l'assembleur LLVM. Cette nouvelle version du programme

binaire est exécutée symboliquement (exécution symbolique dynamique) afin de trouver les différents états du programme et leur prédicat de chemins associés. Les prédicats de chemins sont résolus par un solveur SMT (Z3, de Microsoft, dans ce cas-ci). Le résultat est une entrée au programme qui mènera le programme dans l'état ciblé. Le test final est généré avec une exécution concrète, avec le binaire d'origine, en lui donnant l'entrée générée et en récoltant la sortie. Le couple entrée-sortie est alors sauvegardé comme un nouveau cas de test et intégré à la suite.

Cette approche est pertinente dans le contexte de la correction automatique de travaux. Par contre, le modèle d'exécution symbolique de KLEE date et n'applique pas les nouvelles approches d'exécution symboliques comme *Veritesting*. Aussi, KLEE analyse l'assembleur LLVM et nécessite donc, dans la forme présentée, d'avoir accès au code source des programmes à analyser ou bien de traduire le code objet natif à l'assembleur LLVM.

Validation de MERGEPOINT

MERGEPOINT est un outil d'analyse de programme qui a comme objectif la détection de vulnérabilités logicielles. C'est avec cet outil que les auteurs, dans Avgerinos *et al.* (2014b), ont présenté *Veritesting*. Bien que MERGEPOINT n'a pas comme objectif la génération de tests, les auteurs ont montré l'efficacité de leur outil à vérifier l'entièreté des programmes analysés, et ce, en mesurant la couverture des entrées générées par leur moteur d'exécution symbolique hybride.

CHAPITRE V

PROPOSITION D'UNE STRATÉGIE DE CORRECTION AUTOMATIQUE DE TRAVAUX PRATIQUES BINAIRES

Afin de pouvoir évaluer les travaux étudiants, il faut d'abord connaître le comportement attendu du programme à évaluer. Pour ce faire, nous analysons la solution au travail pratique produite par l'enseignant. Nous en extrayons la liste des blocs de code à exécuter pour couvrir tous les cas traités par le logiciel. Nous générons alors une suite de tests qui fera exécuter chacun de ces blocs de code. À partir de cette suite de tests, nous pouvons analyser les programmes soumis par les étudiants et y trouver les comportements qui diffèrent entre les soumissions étudiantes et le programme de référence. Certains programmes étudiants auront des blocs de code qui ne seront pas couverts par la suite de tests initiale. Cette suite de test sera donc augmentée en générant de nouveaux tests à partir du programme soumis.

5.1 Explicité des comportements

Au coeur de la stratégie proposée se trouve la prémisse que sous une représentation atomique, tout cas de traitement (que nous nommerons comportement) possède au moins un représentant parmi les blocs de bases du programme. Nous basons cette hypothèse sur le fait qu'il est improbable qu'un programme produise un

quelconque résultat sans qu'il y ait des instructions responsables d'exécuter la logique associée. En d'autres mots, il est impossible de trouver un traitement qui n'est pas explicité par au moins un bloc d'instructions. En pratique, nous supposons donc que pour chaque cas distinct de l'exécution de programme, le GFC du programme porte au moins un noeud distinct associé à ce traitement. Il n'y aurait donc pas de logique dans le programme pour laquelle il est impossible d'associer au moins un bloc d'instruction.

5.1.1 Boucles infinies

Plusieurs programmes comportent des boucles qui sont potentiellement infinies. Pensez à une invite de commande par exemple. Ces programmes peuvent porter de la logique qui se produit à chaque itération de boucle, ou une seule fois dans le cas de sortie. Ces cas d'exécutions auront eux aussi au moins un bloc représentant, qui effectue cette logique, dans le GFC du programme. Il n'est donc pas nécessaire d'exécuter ces boucles un nombre potentiellement infini de fois, il suffira simplement de faire exécuter les blocs représentants de ces cas d'exécution.

5.2 Analyse de la solution

Pour tester les différents comportements d'un programme, il faut préalablement les connaître. Nous démarrons donc l'analyse d'un groupe de travaux pratiques par l'analyse du programme solution. Nous cherchons initialement l'ensemble des comportements que le programme présente. Avec cet ensemble découvert, nous tentons de faire produire chacun de ces comportements via différentes entrées au programme. Ultimement, l'ensemble de ces entrées utilisateurs permettent d'exécuter tous les comportements ou fonctions du programme. Nous récoltons également le résultat de l'exécution du programme avec chacune de ces entrées afin d'y

comparer les travaux à évaluer.

5.2.1 Découverte des comportements de la solution

Pour découvrir tous les cas distinctifs de l'exécution d'un programme, nous cherchons donc à découvrir tout le code exécutable d'un programme. Il faut noter que le simulateur PEP/8 simplifie cette hypothèse puisqu'il n'y a pas d'opérations aléatoires ou dépendantes d'autres facteurs que des octets en mémoire et de l'entrée utilisateur. Comme discuté précédemment, le processus de découvrir tout le code exécutable d'un programme binaire est analogue au processus de reconstruction du graphe de flot de contrôle. Dans le graphe de flot de contrôle d'un programme, chaque comportement à découvrir est représenté par un noeud. L'exécution de tous les comportements est analogue à l'exécution de tous les blocs de code représentés par des noeuds dans le GFC du programme.

Une fois tous les comportements (noeuds du GFC) identifiés, nous cherchons à les faire exécuter. L'exécution symbolique dynamique permet d'exécuter un programme en récoltant les contraintes nécessaires à la production des états possibles de l'exécution d'un programme. Nous pouvons donc cibler un des noeuds du graphe du flot de contrôle, exécuter symboliquement le programme afin de connaître les contraintes nécessaires à l'exécution du bloc de code, puis utiliser l'ensemble de contraintes résultant pour générer une entrée au programme. Le programme, exécuté avec cette entrée, exécutera le bloc de code ciblé. À partir de l'entrée qui fera en sorte d'exécuter le bloc ciblé, nous devons continuer l'exécution jusqu'à la terminaison du programme pour observer le résultat du programme. En forçant le prédicat de chemin trouvé pour atteindre le bloc ciblé et en continuant l'exécution symbolique pour trouver les contraintes additionnelles pour atteindre une des fins du programme, on trouve un prédicat qui exprime l'exécution du bloc ciblé puis

la terminaison du programme. Une solution à ce prédicat est donc une entrée au programme qui fera en sorte d'exécuter le bloc ciblé et puis de terminer l'exécution du programme. Nous pouvons ensuite répéter ce processus pour tous les blocs du GFC.

L'ensemble de contraintes recueillies lors de l'exécution symbolique représente les différents chemins de l'exécution du programme en fonction des entrées qui lui sont données. Pour représenter l'exécution d'un bloc en particulier, nous pouvons insérer une contrainte de plus qui contraint le pointeur d'instruction à contenir l'adresse de la première instruction du bloc ciblé pour au moins un état de sa trace d'exécution. Avec l'ajout de cette contrainte, toute solution au système d'équations analogue à l'ensemble de contraintes assurera l'exécution du bloc de code ciblé. Une solution à ce système présente donc un ensemble de valeurs possibles des cases mémoires ou entrées au programme lors de l'état initial (avant l'exécution de la première instruction du programme) qui mène le programme dans l'état ciblé. De cette façon, nous sommes en mesure de trouver les entrées utilisateurs nécessaires à faire exécuter un bloc de code ciblé. Il suffit donc de tenter de résoudre les contraintes recueillies pour exécuter chacun des blocs identifiés lors de l'étape précédente. L'exécution symbolique permet également de trouver une solution pour les valeurs de sorties du programme. Celles-ci seront validées à l'étape suivante.

Avec les entrées nécessaires à l'exécution de chaque bloc identifié, nous pouvons valider que chacune fait bien exécuter le code ciblé. Une machine virtuelle PEP/8 faiblement instrumentée est employée. Elle exécute concrètement le programme analysé avec les entrées données et trace les valeurs du pointeur d'instruction. Celà permet de vérifier que l'adresse du bloc ciblé se retrouve dans la trace, nous confirmant l'exécution du bloc. Cette même exécution concrète permet d'obtenir les sorties réelles du programme pour les entrées données et sauvegarder le

couple entrées/sorties pour une utilisation lors de la correction des soumissions étudiantes. Il est également possible, à cette étape, de valider les sorties attendues du cas de test généré. Des sorties générées qui, lors de la résolution des contraintes issues de l'exécution symbolique sont différentes de celles produites lors de l'exécution concrète pour les mêmes entrées, sont un bon indicateur qu'une perte de précision a eu lieu lors de l'analyse du programme.

5.3 Analyse de la soumission

La phase d'analyse de la soumission d'un étudiant commence de la même façon que l'analyse de la solution du professeur. Initialement, nous générons le GFC du programme soumis. Celui-ci nous permettra de connaître les différents blocs de code présents dans le programme. Une fois tous les blocs identifiés, nous sommes en mesure d'analyser la couverture du code faite par la suite de tests que nous avons générée lors de l'analyse du programme solution.

La machine virtuelle instrumentée discutée précédemment nous permettra d'accumuler quels blocs de code, et donc quels noeuds du GFC sont exécutés lors de l'exécution de la suite de tests. Ceci nous informe également sur les noeuds du GFC qui n'ont pas été exécutés par le programme en cours d'évaluation. Nous répétons alors le même cheminement que pour générer les tests initiaux, mais cette fois-ci pour chacun des blocs de la soumission qui n'aura pas été exécuté par la suite de tests. Celà génèrera une suite de test propre à chaque soumission, mais qui ne correspond à aucun bloc en particulier dans le programme solution. Ces tests représentent des cas qui sont traités dans la soumission analysée, mais qui n'ont pas de logique particulière dans le programme solution. Ces nouveaux cas de tests sont exécutés concrètement contre le programme solution et la soumission analysée et les résultats sont comparés. Cette passe additionnelle de test

permet de valider si ces blocs de code induisent un comportement déviant dans le programme de l'étudiant.

5.4 Comparaison des résultats de tests

Suite à ces passes d'exécutions symboliques dynamiques et concrètes, une suite de tests couvrante a été générée tant sur le programme de référence que sur chacune des soumissions étudiantes, la première partie étant la suite de tests commune (générée en analysant le programme solution), l'autre celle issue de l'analyse des programmes soumis.

Il est alors possible d'identifier les comportements déviants de l'application soumise par l'étudiant en comparant les sorties de chaque cas de test aux sorties attendues, soient celles générées par le programme solution.

5.5 Exemple d'application

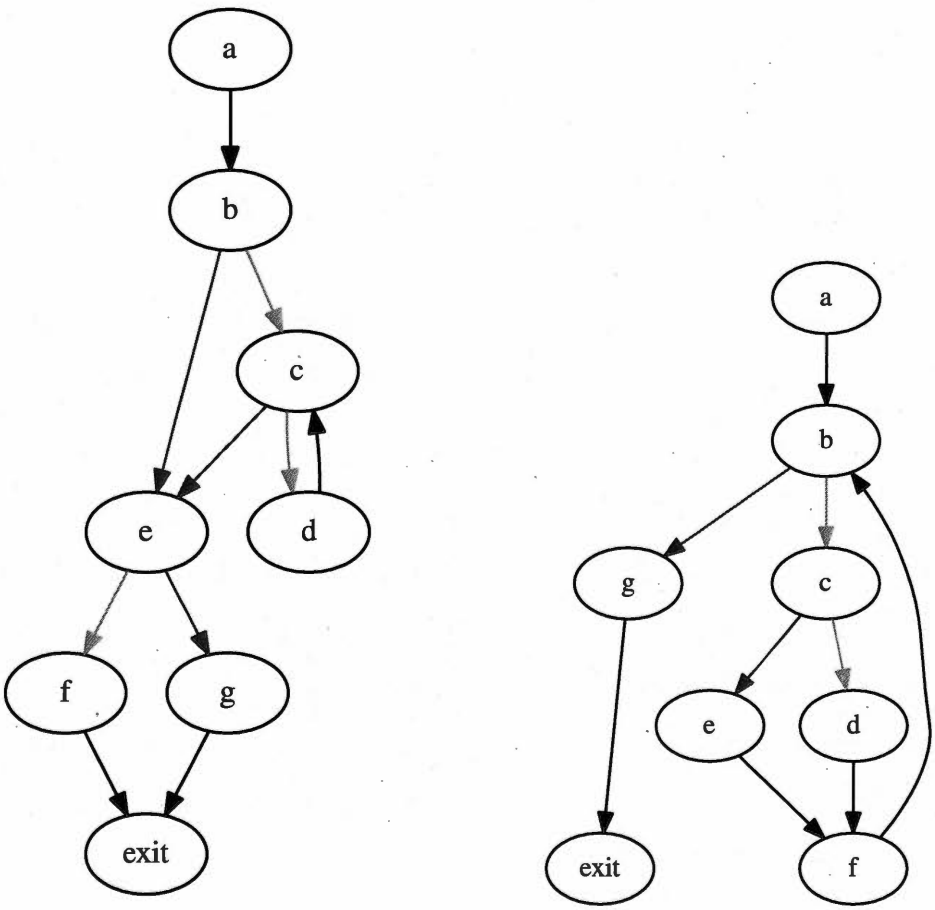
Pour visualiser la stratégie, cette section présente les phases de la génération de tests pour un travail pratique fictif. La Figure 5.1 présente les graphes de flots de contrôle de deux programmes. En (a) on trouve le GFC du programme de référence et en (b) celui d'un programme soumis par un étudiant.

La stratégie commence toujours par la reconstruction du graphe de flot de contrôle de la solution (Figure 5.1 (a)). À partir de ce graphe, nous ciblons, un à la fois, les noeuds du graphe qui restent à couvrir par un test. Nous générons une entrée au programme qui fera exécuter les instructions du noeud ciblé, puis récoltons le couple entrée/sortie en exécutant concrètement le programme solution avec l'entrée générée. Cette exécution concrète permet également de recueillir la trace des blocs exécutés. Cette phase de génération de tests est présentée en Figure 5.2.

Les graphes (a) jusqu'à (d) montrent le bloc ciblé en gras, puis la trace de blocs exécutés par l'entrée générée. En (e), nous illustrons la couverture totale faite par la suite de tests en cumulant les tests de (a) à (d).

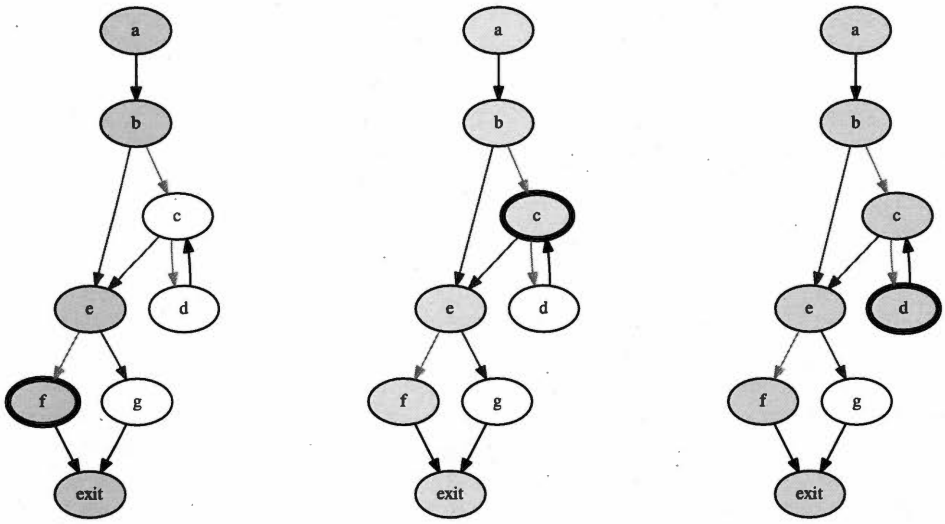
La phase de traitement de la soumission étudiante commence aussi par la reconstruction du graphe de flot de contrôle du programme (Figure 5.1 (b)). Ensuite, la suite de tests issue de l'analyse du programme solution est exécutée concrètement sur la soumission. Nous en extrayons la trace des blocs couverts par chaque test (Figure 5.3 (a)). Nous pouvons alors cibler les blocs du programme soumis qui ne sont pas couverts par la suite de tests commune et générer des entrées qui feront exécuter ces blocs. En Figure 5.3 (b), nous montrons l'augmentation de la suite de tests pour couvrir le noeud d de la soumission, puis en (c) la couverture de la suite de tests augmentée.

Par la suite, l'exécution concrète de la suite de tests augmentée sur la solution permet d'obtenir les comportements de souhaités pour chaque cas traité, tant dans la solution que dans la soumission d'un étudiant. Nous pouvons alors exécuter concrètement cette suite de tests sur la soumission étudiante et comparer les couples entrée/sortie respectifs de la suite de test exécutée sur la solution et constater les comportements erronés du travail soumis.

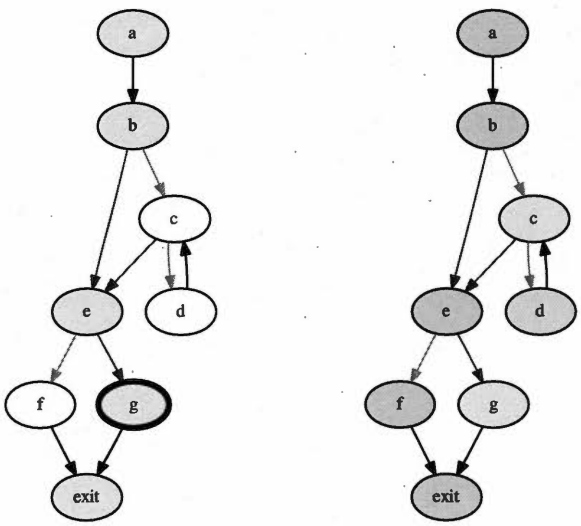


(a) (b)

Figure 5.1: GFC du programme solution et d'une soumission étudiante.

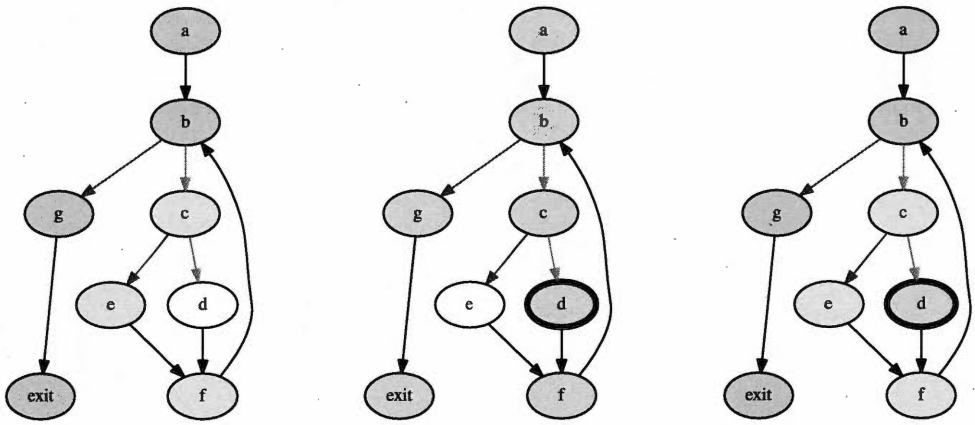


(a) (b) (c)



(d) (e)

Figure 5.2: Couverture de blocs lors de la construction de la suite de tests.



(a) (b) (c)

Figure 5.3: Couverture de blocs sur la soumission.

CHAPITRE VI

IMPLÉMENTATION

6.1 Choix d'architecture et extension d'ANGR

Deux stratégies d'implémentation ont été envisagées : étendre une suite d'analyse existante ou bien en développer une nouvelle spécifique au simulateur de processeur PEP/8. Les deux approches présentaient leur lot d'avantages et d'inconvénients, mais ultimement, l'approche d'étendre une suite existante offrait des avantages considérables dans le cadre d'un travail de maîtrise.

Le plus grand avantage de la stratégie choisie est qu'une grande quantité d'effort a déjà été investie par la communauté de ces projets afin d'y intégrer de nombreux composants réutilisables, le tout dans le but ultime de réduire le travail d'implémentation répété des différentes analyses, stratégies et modélisations de la littérature scientifique. (Shoshitaishvili *et al.*, 2016) Par exemple, ANGR propose déjà plusieurs analyses ainsi que des algorithmes documentés de reconstruction de graphes de flots de contrôle, de découverte de variables et d'exécution symbolique. Utiliser les mêmes algorithmes et implémentations que d'autres chercheurs offre aussi l'avantage d'une meilleure comparaison. Les détails des technologies et des implémentations sous-jacentes peuvent être abstraits ce qui permet donc de comparer adéquatement la nouvelle contribution. Les suites existantes ne supportant pas initialement PEP/8, mais plutôt des architectures communes (*x86*,

ARM, etc.), la stratégie d'étendre une suite existante assure une certaine proximité avec les outils et stratégies d'analyse de code conçus pour des architectures non pédagogiques. C'est une façon de s'assurer que les techniques développées restent réalistes sur une architecture plus complexe et ne deviendrait pas spécifiques au simulateur PEP/8. De plus, cette stratégie laisse la porte ouverte au support de travaux pratiques sur d'autres architectures, et ce sans réécriture complète du projet d'analyse de travaux pratiques.

Par contre, utiliser une suite d'analyse existante signifie également utiliser une suite qui n'est pas spécifique au simulateur de processeur PEP/8. Aucune des spécificités du simulateur n'y sera préalablement modélisée et l'inclusion de ces spécificités, sans nuire aux décisions d'architecture prises pour supporter d'autres architectures de processeurs, sera possiblement complexe. Par le fait même d'inclure une suite complète d'analyse au projet, le travail de compréhension en profondeur du fonctionnement de plusieurs composants de la suite est ajouté à la tâche d'implémentation. Bien que l'exercice se veut particulièrement formateur, le fardeau qu'il représente est substantiel. Un dernier point considéré lors du choix de cette stratégie est la nature changeante d'un projet logiciel libre à source ouvert et à évolution rapide. Il faudra considérer que les stratégies développées risquent de devoir être adaptées et réintégrées plusieurs fois à la base de code vu le rythme de développement des outils d'analyses libres disponibles.

6.2 Niveaux d'analyses

L'implémentation de la solution proposée se divise en deux grandes sections. Une première qui présente les extensions nécessaires à permettre l'analyse de programme PEP/8 au travers d'ANGR. Une seconde qui détaille les mécanismes d'analyse automatique de travaux pratiques. On y retrouve les notions propres

aux travaux étudiants sur lesquels se concentre la solution.

6.2.1 Implémentation des composants d'extension

L'intégration d'une nouvelle architecture à ANGR nécessite de développer plusieurs composants qui permettront au cadriciel d'analyser un programme de cette architecture normalement.

Un module de *lifting* est responsable de représenter les octets de code binaire en VEX, une représentation abstraite qui offre plusieurs avantages pour l'analyse, particulièrement l'explicité de sa représentation et son abstraction de différentes architectures réelles.

ANGR travaille également avec une modélisation particulière de chaque architecture et spécification de processeur qu'il est en mesure d'analyser. Cette modélisation est utile lors de plusieurs analyses puisqu'elle introduit différentes spécifications du processeur comme le nombre de bits, la taille des mots ou encore les conventions d'appels et d'utilisation des registres.

L'analyse symbolique naïve des différents appels systèmes exposés par un système d'exploitation, dût à leur fréquente complexité, est souvent prône à plusieurs difficultés. Il est donc souvent préférable de simuler les appels système complexes. Ceci implique de réimplémenter leur comportement, exact ou simplifié, et d'y inclure une logique symbolique.

Une dernière composante est nécessaire afin de pouvoir analyser un programme binaire : un chargeur. Bien qu'ANGR expose déjà un module de chargement mémoire, il faut l'étendre pour supporter les spécificités de l'architecture ajoutée. En plus d'interpréter les segments mémoires, d'exposer la logique de chargement du programme et de ses bibliothèques en mémoire et de gérer l'initialisation de l'exé-

cution, le chargeur doit charger et gérer les différents *hooks* et rappels (*callbacks*) qui sont introduits par les autres modules.

Lifter

Le rôle principal du module de *lifting* est d'expliciter clairement les effets de chaque instruction machine (Shoshitaishvili *et al.*, 2016). Par exemple, une instruction d'empilage (`PUSH`) a plusieurs effets : l'écriture d'une valeur en mémoire et le décalage du pointeur de pile. L'instruction sera donc ramenée à une représentation intermédiaire où chaque effet sera explicité en une instruction du langage intermédiaire. Un autre avantage considérable d'une représentation intermédiaire de cette forme est qu'il est facile de la transformer en forme d'assignation simple statique (ASS ou *Single Static Assignment*, *SSA*), une forme qui facilite plusieurs analyses (Cytron *et al.*, 1989). Un dernier avantage d'effectuer les analyses sur une représentation intermédiaire est d'abstraire les particularités des différentes architectures. Celà permet au moteur d'analyser de façon analogue les programmes de différentes architectures.

Les auteurs d'ANGR ont choisi VEX, la représentation intermédiaire (RI) développée par le projet VALGRIND comme représentation intermédiaire pour plusieurs raisons. Principalement, le moteur de *lifting* étant déjà existant pour toutes les architectures intéressant initialement le projet, à l'exception du format binaire du *Darpa Cyber Grand Challenge*, donc les auteurs pouvaient se concentrer sur l'écriture du moteur d'analyse plutôt qu'au développement d'un *lifter*. VEX est une représentation de la forme SSA, mais où il est possible de représenter les instructions en forme arborescente. Cette forme peut ensuite être facilement aplatie si nécessaire. Le projet VALGRIND avait également inclus la possibilité d'instrumenter la RI générée, ce qui est intéressant pour plusieurs analyses dynamiques.

ANGR manipule la représentation intermédiaire VEX sous sa forme arborescente et en mémoire (par opposition à traiter la représentation textuelle aplatie). Le tout se fait au travers d'une interface de fonctions étrangères avec le langage C, dans lequel est écrit le module de *lifting* original. PyVEX est le module de la suite qui expose une interface en PYTHON de manipulation et de génération de structures (et de représentation) VEX.

Afin d'être en mesure de représenter un programme PEP/8 en VEX sous sa forme arborescente en mémoire, il faut générer l'arborescence de structures C qui représente le programme. Pour ce faire, il a fallu écrire un tout nouveau module de *lifting* qui transforme chaque instruction PEP/8 codée en format binaire en sa représentation intermédiaire VEX. Le tout a été écrit en langage PYTHON au travers de l'API qu'expose PyVEX. Le nouveau module manipule donc directement la représentation VEX en mémoire. La seule différence avec le traitement d'un programme *x86*, initialement supporté par ANGR, est que la phase de génération d'IR est exécutée au travers de PyVEX (qui utilise la FFI avec C de PYTHON).

La phase de *lifting* commence par le désassemblage de l'instruction visée en mémoire. Une fois l'instruction décodée, elle doit être traduite en une ou plusieurs (normalement plusieurs) instructions VEX. Pour chaque instruction VEX, la structure arborescente VEX doit être générée en mémoire. Le découpage en opérations atomiques doit être effectué pour chaque instruction décodée. Par exemple, une instruction `LDA 10, i` en PEP/8 peut être traduite en plusieurs instructions VEX.

```
LDA 10,i
```

Listing 6.1: Une instruction PEP/8.

```
iIRSB {
   t0:Ity_I16 t1:Ity_I16 t2:Ity_I16 t3:Ity_I1 t4:Ity_I16

   00 | ------ IMark(0x0, 3, 0) ------
   01 | t0 = LDbe:I16(0x0001)
   02 | PUT(a) = t0
   03 | t1 = Shr16(t0,0x0f)
   04 | t2 = And16(0x0001,t1)
   05 | PUT(n) = t2
   06 | t3 = CmpEQ16(t0,0x0000)
   07 | t4 = 1Uto16(t3)
   08 | PUT(z) = t4
   NEXT: PUT(ip) = None; Ijk_NoDecode
}
```

Listing 6.2: `LDA 10,i` transcrit en VEX.

Dans le Listing 6.2, on voit en prologue la déclaration des variables temporaires ainsi que leur type (entiers 1 ou 16 bits). En ligne `00`, le IMark indique l'adresse (`0x00`) du bloc lifté et sa longueur en octets (3). Ensuite la liste des instructions VEX, suivie du bloc successeur potentiel (dans ce cas il n'y en a pas). On voit donc le chargement (`LDbe`) de la case mémoire de l'opérande (qui est à l'adresse `0x0001`), l'écriture dans la case mémoire correspondant au registre `A` ainsi que les lectures en mémoire et les opérations mathématiques nécessaires aux calculs des registres d'état ainsi que leur sauvegarde.

Plusieurs complexités additionnelles apparaissent lorsque l'on veut analyser du code mutant. Le décodage des instructions d'un bloc ne peut se faire un bloc

à la fois dès qu'il y a une écriture dans une case mémoire incluse dans le bloc d'instruction. Il faut plutôt, à chaque instruction à exécuter, la charger de la mémoire immédiatement avant de l'exécuter, au cas où l'instruction précédente l'aurait modifiée. De plus, si l'instruction elle-même est symbolique, il faudra gérer l'éventualité de toutes les instructions possibles qui pourraient la remplacer. Souvent, la solution pratique qui est employée est de concrétiser l'instruction, c'est-à-dire, trouver une stratégie qui limite les valeurs possibles de l'instruction afin de contenir l'explosion combinatoire potentielle.

Modélisation de l'architecture du processeur

Plusieurs analyses utilisées dans ANGR ont un gain d'efficacité à connaitre les spécificités de l'architecture du programme analysé (Shellphish, 2018). C'est pourquoi le cadriciel modélise une bonne partie de la spécification et des conventions des architectures processeurs supportées. Les spécifications sur le boutisme, les registres et les conventions d'appels ou d'utilisation de registres y seront, entre autres, modélisées. Un exemple de convention modélisée est si l'instruction d'appel de sous-programme (`CALL`) empile ou non une adresse de retour.

Pour des raisons de simplicité, l'adressage des registres est identique à l'adressage de la mémoire. En effet, au niveau de VEX, les registres sont spécifiés dans la représentation du processeur comme des adresses dans l'espace adressable du programme. ANGR garde également une table d'association entre les registres, leur adresse en mémoire, et pour l'aspect pratique, leur nom.

Les conventions qui y sont modélisées aident certaines analyses à préciser leur résultat ou à réduire le niveau d'imprécision de leurs résultats. Les analyses de découverte de code, par exemple, profitent grandement de connaître l'allure d'un prologue et d'un épilogue de fonction normal. Ces informations sont aussi utiles

comme heuristique dans le cas de résolution de destinations de **CALL** symboliques. Les conventions d'appels aident à l'identification des arguments et des variables locales lors de l'appel de sous-programmes.

Dans le cas de l'architecture PEP/8, l'élément clé de la modélisation des conventions est en fait d'étendre le comportement de certaines analyses afin de supporter une architecture aussi peu spécifiée. En effet, en PEP/8, il n'y a pas de conventions particulières pour beaucoup d'instructions ou de routines qui sont normalement spécifiées. Par exemple, il n'y a aucune convention d'appel pour les sous-programmes. L'appelant comme l'appelé peuvent être responsables de la gestion du cadre d'appel. Il peut aussi n'y avoir aucun cadre d'appel du tout. Un autre exemple est le cas de la recherche de fonctions dans un programme. Les modélisations habituelles d'ANGR représentent l'encodage normal d'un prologue et d'un épilogue de fonction. Dans le cas de PEP/8, ces deux normes sont inexistantes. Les analyses doivent donc se fier sur la recherche d'instructions **CALL** et **RET**, toutes deux codées sur un seul octet. Celà peut mener à beaucoup de bruit dans l'identification des frontières des fonctions et, éventuellement, ces erreurs se feront ressentir au travers de faux-positifs additionnels.

Système d'exploitation et procédures simulées

Les appels système ainsi que les appels à des bibliothèques partagées ont souvent été problématiques pour les outils d'analyse de code. Parfois à cause de la complexité (ou même la taille) du code qui y est présent. D'autre fois à cause de l'effort d'implémentation que représente de supporter l'entièreté du code exposé. Une approche traditionnelle à ce problème est d'abstraire le système d'exploitation et d'exposer des appels système simplifiés (Cadar *et al.*, 2008; Li *et al.*, 2011; Cha *et al.*, 2012; Avgerinos *et al.*, 2014a,b; Shoshitaishvili *et al.*, 2016). Cette

approche limite la complexité d'analyse pour ces appels. Dans le cas d'un outil d'analyse symbolique, l'abstraction est également utile pour utiliser des raccourcis dans l'exécution symbolique (concrétiser un mode d'adressage par exemple). Ceci permet également de court-circuiter les appels trop difficiles à analyser naïvement (au niveau binaire, comme le reste du code du programme) et d'introduire une version exposant un comportement limité. Bien que cela implique une perte de précision ou de complétude (ou les deux) dépendamment de la version introduite, cette stratégie permet d'analyser dans un temps raisonnable (voire de rendre possible l'analyse de) certains programmes utilisant dès appels de fonctions ou des appels système trop complexes. Des exemples de fonctions complexes pertinentes à abstraire sont les fonctions de chaînes à formats exposés par la `libc` comme `scanf` .

L'architecture proposée par ANGR pour répondre au besoin d'abstraction des appels systèmes et des bibliothèques est d'implémenter ces fonctions directement en PYTHON et de les substituer aux appels qu'ils modélisent et d'exécuter la nouvelle logique plutôt que le code binaire correspondant. L'implémentation en PYTHON manipule directement le modèle d'état du programme et peut donc faire n'importe quelles manipulations sur les représentations concrètes et symboliques de l'état de celui-ci.

Il faut noter qu'une implémentation en PYTHON d'une procédure a un désavantage considérable dans le cas de code mutant : la procédure étant réimplémentée et substituée à l'originale, il devient extrêmement difficile de faire suivre les changements que le programme apporterait à son propre code sur l'engin d'exécution symbolique. Comme la procédure au format binaire n'est plus utilisée et donc pas analysée, si le programme tente de la modifier en changeant ses instructions en mémoire, refléter ces changements dans une autre zone mémoire, sur une implémentation d'un autre langage, et les appliquer correctement pour la suite de

l'exécution symbolique est très complexe.

Dans le cas de l'implémentation de cette abstraction pour l'analyse de travaux étudiants en PEP/8, il aurait été également possible de ne pas faire de cas spéciaux pour les appels système autre que de gérer le comportement des interruptions, puisque tout le code du noyau PEP/8 est préalablement chargé en mémoire à une adresse fixe.

Les deux approches ont leurs avantages et leurs inconvénients. Le choix d'implémenter une abstraction du système d'exploitation était motivé par deux raisons en particulier. Premièrement, la simulation des appels système d'entrées/sorties permet de standardiser la gestion des flux d'entrées/sorties avec l'utilisation normale d'ANGR. En résultat, l'utilisation de ces flux dans les analyses à haut-niveau est simplifiée et enrichie. Un autre avantage de l'ajout d'une abstraction du système d'exploitation est de respecter la philosophie des outils d'analyse de code (principalement ANGR) et d'explorer et de comprendre le rôle et le fonctionnement de cette modélisation. Le désavantage le plus important est l'impossibilité de gérer les interruptions personnalisées (via les interruptions libres `NOPn`) de la même façon que les autres appels systèmes exposés par le noyau PEP/8.

Modélisation de la logique de chargement

Le dernier module implémenté au niveau d'ANGR est le module de simulation de chargement en mémoire d'un programme. Plus spécifiquement, dans le cas d'un chargeur PEP/8, le rôle de ce module est plutôt restreint puisque le chargement d'un programme PEP/8 est simple. Notamment, il n'y a aucune bibliothèque partagée à charger et donc aucune logique de chargement dynamique à prévoir ou à gérer. Les seules choses à charger en mémoire sont les instructions du programme lui-même ainsi que le noyau du système d'exploitation PEP/8. Les programmes

PEP/8 n'ont également pas de sections mémoire à gérer ni différents segments mémoire.

Bien que cette structure simplifiée de la mémoire rend la tâche de modéliser le chargement d'un programme en mémoire moins complexe, quelques changements à ANGR sont nécessaires afin de supporter des programmes avec une structure de la mémoire aussi simple (sans bibliothèques ni section).

6.2.2 Implémentation du moteur d'analyse de travaux pratiques

ANGR étant un cadriciel en PYTHON, le moteur d'analyse de travaux pratiques est également écrit en PYTHON. La reconstruction du graphe de flot de contrôle des programmes *PEP/8* est réalisé par le module `CFGAccurate` d'ANGR. Celui-ci reconstruit le GFC en alternant entre des phases de découverte itérative de blocs et d'exécution symbolique dynamique partielle pour la résolution des branchements indirects. Le segment de code exécuté symboliquement est choisi en traversant le GFC à reculons pour un nombre empiriquement choisi de noeuds (8) en partant du branchement indirect. Pour déterminer comment diriger l'exécution des programmes vers les blocs ciblés, nous avons utilisé le moteur d'exécution symbolique dynamique d'ANGR, avec l'extension *Veritesting*, ainsi qu'un heuristique de priorisation de chemins. L'heuristique profite du GFC pour élaguer les exécuteurs des branches qui ne mènent pas au bloc ciblé. Dans ANGR, cette stratégie de recherche est disponible dans le module `explorer`.

CHAPITRE VII

ÉVALUATION

7.1 Rappel des objectifs

L'objectif initial de parvenir à corriger automatiquement des travaux étudiants peut se traduire en recherche de comportements déviants (ou absents) dans le travail soumis par un étudiant en le comparant à une solution fonctionnelle fournie par le professeur. On cherche également à être en mesure de donner à l'étudiant un cas de test (exemple d'entrées) qui permettra de témoigner du comportement déviant (et donc de le déclencher).

7.1.1 Corpus de travaux pratiques

Le corpus compte deux énoncés de travaux pratiques différents étalés sur deux sessions pour un total de 81 remises à corriger. Afin de faciliter la récolte de statistiques, nous avons également accès au code source (en langage d'assemblage) de chaque programme. En guise de rappel, le code source de ces programmes n'a pas été utilisé lors de leur analyse. Le code source a donc servi uniquement à la récolte de statistiques afin d'évaluer la stratégie de correction automatique.

Statistique	TP-A15-MORSE	TP-H16-CALC	Global
Pourcentage de tests valides (ESD versus Exécution Concrète)	43.58%	17.64%	31.69%
Pourcentage des instructions assemblées exécutées	72.13%	58.47%	65.87%
Pourcentage de blocs du GFC exécutés	98.38%	100.00%	99.12%

Tableau 7.1: Résumé des résultats de l'expérimentation

7.1.2 Conditions d'évaluation

Toutes les expérimentations ont été faites sous les mêmes contraintes matérielles et temporelles. Notre outil disposait d'un maximum de 15 minutes par travail pratique à analyser sur une machine avec un processeur à 3.10 GHz Intel (R) Core i7 4510U et de 4Gb de mémoire vive.

7.2 Mesures

7.2.1 Complétion de l'analyse

Avec les contraintes mentionnées en sous-section 7.1.2, notre outil a pu obtenir des résultats sur 47 des 81 programmes étudiants (56%). Nous avons investigué la raison des échecs de l'analyse sur les travaux n'ayant pu être analysés. Pour quelques-uns, notre outil était en mesure d'obtenir des résultats si nous retirions la limite de temps à l'analyse. Dans ces cas, le ralentissement était attribuable à l'explosion combinatoire du nombre de chemins dans l'ESD et qui remplissaient la mémoire vive de la machine de test. Pour d'autres, les prédicats de chemins devenaient assez complexes pour que le solveur (Z3) soit incapable de générer une solution par manque de mémoire.

Le reste des statistiques recueillies ne considèrent que les programmes pour lesquels l'outil a été en mesure de produire un résultat.

7.2.2 Validité des cas de tests

Comme mentionné précédemment (chapitre 2), les analyses de programmes peuvent potentiellement produire des résultats erronés. Ceux-ci sont normalement attribuables à certaines approximations des stratégies théoriques d'analyse de permettre la complétion de celle-ci dans un délai raisonnable, avec des ressources raisonnables (ou une combinaison des deux).

L'exactitude des cas de test est plutôt simple à valider. Nous n'avons qu'à exécuter concrètement le programme analysé avec les intrants spécifiés par le cas de test et nous assurer de la concordance des deux résultats. En d'autres mots, s'assurer que les entrées/sorties générées comme cas de test lors de l'exécution symbolique dynamique sont les mêmes qu'à l'exécution concrète.

Le ratio de cas de tests valides par rapport au nombre de cas de test générés éclaire sur le niveau d'approximations qu'il a fallu faire afin de parvenir à obtenir des résultats d'analyse sur le programme ciblé.

La première ligne du Tableau 7.1 présente le taux de cas de tests dont la solution au prédicat de chemin issue de l'exécution symbolique concorde avec le résultat de l'exécution concrète. Le taux important de différence entre les résultats des exécutions symboliques dynamiques et concrètes est attribuable à deux facteurs : des erreurs liées aux approximations faites pendant l'analyse, et des erreurs (bogues) dans l'implémentation de la solution d'analyse. Il est donc évident que même pour une architecture simple comme celle du simulateur *PEP/8*, le niveau d'approximations effectuées lors des analyses de programmes binaires est un facteur important des résultats rendus.

Il est à noter que bien que les résultats attendus des tests générés diffèrent souvent des résultats de leurs exécutions concrètes, il en résulte une suite de tests tout de

même valides puisque seuls les résultats des exécutions concrètes (de la solution et de la soumission) sont ultimement utilisés. En d'autres mots, les sorties attendues extraites des passes d'analyse symbolique ne sont pas utilisés dans la génération de tests. Les entrées évaluées sont utilisées pour alimenter une exécution concrète. Nous utilisons ensuite la sortie de cette exécution concrète comme sortie attendue des tests générés. Cette mesure n'a donc pas d'impact direct sur l'identification de comportements déviants.

7.2.3 Découverte et couverture de code

Afin de déterminer si la solution proposée est adéquatement couvrante sur le logiciel analysé, nous devons d'abord déterminer le taux de code qui est exécuté par la suite de tests générée pour chaque programme.

Le calcul du taux de code exécuté par rapport au code réel nécessite de connaître ce qui représente l'ensemble du code à découvrir. Comme le corpus de travaux étudiants avec lequel nous travaillons a également été livré sous la forme de langage d'assemblage, nous sommes en mesure d'identifier l'ensemble des instructions dans le programme. Évidemment, cette démarche considère que les travaux soumis ne font pas utilisation de code mutant. Afin d'extraire automatiquement les ensembles réels d'instructions de chaque programme, nous avons implémenté un assembleur PEP/8 légèrement instrumenté, qui permet d'extraire la liste des adresses de toutes les instructions assemblées (en ignorant les directives). Un bémol à considérer : le code mort présent dans le code source du programme sera considéré dans la liste des instructions à potentiellement exécuter. Par contre, le GFC, naturellement, ne portera pas de blocs correspondant, puisque ces instructions sont inatteignables.

Les résultats du Tableau 7.1 montrent qu'il y a certainement un manque à gagner

dans la couverture des instructions du programme. Bien qu'une partie des instructions non exécutées est attribuable au code mort dans les travaux des étudiants, il est difficile de croire que 34% du code remis soit mort.

Atteinte des noeuds du GFC

Il est intéressant de vérifier si les manquements en découverte de code sont attribuables à la phase de reconstruction du graphe de flot de contrôle ou plutôt à la phase d'exécution symbolique du programme. Comme mentionné précédemment, plusieurs types d'instructions peuvent être problématiques dans la phase de reconstruction du GFC. Ce sont ce genre d'embuches qui nuiront à une reconstruction complète et correcte du GFC. Comme l'ensemble des instructions considérées comme potentiellement exécutables sont représentées dans le GFC, si celui-ci est incomplet, nous serons incapables de chercher à exécuter les blocs de codes qui n'auront pas été découverts.

Pour nous éclairer sur la situation, nous avons récolté les blocs du GFC qui ont été visités lors de l'exécution concrète de la suite de tests générée pour chaque programme.

Les résultats obtenus montrent clairement que la phase d'exécution symbolique parvient bien à visiter la quasi-totalité des blocs du GFC même avec les contraintes mentionnées en sous-section 7.1.2. Nous pouvons donc attribuer la couverture incomplète des instructions du programme à la phase de reconstruction du graphe de flot de contrôle.

CONCLUSION

Les récentes avancées en analyse des programmes binaires, principalement dans le domaine de l'exploitation automatique de vulnérabilités, nous ont menées à nous poser la question : «Est-il possible d'appliquer ces nouvelles techniques de pointe en analyse de programmes binaires à la correction de travaux pratiques dans un cadre académique ? »

Nous avons alors cherché à appliquer les techniques avancées par le domaine de l'exploitation automatique aux travaux pratiques du cours INF2170 : *Organisation des ordinateurs et assembleur* de l'Université du Québec à Montréal. Dans ce cours, les étudiants étudient le comportement d'un ordinateur au travers d'un simulateur de processeur : *PEP/8*. Le simulateur exécute des programmes sur son propre jeu d'instruction ; qui est réduit, mais réaliste. Les travaux pratiques de ce cours s'avèrent donc pertinents à étudier dans le cadre de l'analyse de programmes binaires.

Nous avons choisi d'attaquer le problème de la découverte des comportements déviants des soumissions des étudiants par rapport aux comportements du travail de référence écrit par l'enseignant. Ce choix élimine le besoin de développer manuellement une suite de tests couvrant approximativement les cas à traiter par le programme et le besoin d'analyser manuellement chaque travail pratique afin de tenter d'y identifier tous les comportements incorrects qui n'auraient pas étés couverts par la suite de test.

Dans ce travail, nous proposons donc une solution de correction automatique de travaux pratiques binaires. La stratégie proposée tente d'identifier les compor-

tements déviants (ou absents) dans les programmes soumis par les étudiants en les comparant à une solution fonctionnelle fournie par le professeur. La solution proposée est également en mesure de fournir des cas de test qui permettront aux étudiants de déclencher les comportements déviants.

La solution se base sur un graphe de flot de contrôle reconstruit à partir du programme binaire afin de guider la suite de l'analyse du programme. En faisant le lien entre les différents traitements distinct d'un programme et les noeuds dans le graphe de flot de contrôle, nous sommes en mesure d'identifier les différents cas traités par le programme.

Avec notre stratégie basée sur l'exécution symbolique dynamique augmentée avec *Veritesting* et aidée d'une heuristique de priorisation de chemins, nous cherchons à déclencher l'exécution de tous les cas de traitements distincts représentés par les blocs de base du programme. Pour ce faire, nous trouvons une solution au prédicat de chemin de chaque bloc de code à cibler. Par le fait même de résoudre le prédicat de chemin des blocs ciblés, nous obtenons un représentant des entrées au programme qui déclenchera le traitement effectué par ces blocs. Cette entrée, jumelée au résultat d'une exécution concrète du programme avec celle-ci, constitue un cas de test qui permet de valider ce traitement.

En employant cette approche de génération de tests, nous pouvons générer des suites de tests capables de couvrir le code du programme solution ainsi que des programmes soumis par les étudiants. Ces suites de tests peuvent alors être utilisées pour détecter automatiquement les comportements déviants des travaux des étudiants.

Notre implémentation de la solution a démontré la pertinence de la stratégie proposée. Elle a non seulement permis de réaliser qu'il était envisageable d'employer des stratégies d'analyse de programme binaires à la correction automatique de

travaux, mais également de voir quelles analyses présentent un manque à gagner intéressant.

Bien que nous avons démontré l'intérêt des techniques de pointe en analyse de programmes binaires pour la correction de travaux dans un cadre académique, la solution que nous avons proposée ne constitue pas une finalité pour la problématique. Nous nous demandons notamment si l'utilisation du prédicat de chemin lui-même, plutôt qu'une solution à celui-ci, peut être utilisée adéquatement comme entrée-test symbolique à un autre programme respectant la même spécification, comme dans le cas des travaux pratiques.

D'autres techniques d'analyse de programmes sont également applicables à la découverte et au déclenchement d'états dans un programme comme le *fuzzing*, et encore plus particulièrement le *fuzzing* symboliquement assisté (Stephens *et al.*, 2016), une approche qui profite d'un moteur d'exécution symbolique pour débloquer le moteur de génération d'entrées lorsque celui peine à déclencher de nouveaux états.

Une autre tangente de recherche nous paraît pertinente : l'identification de mauvaises pratiques de programmation. Un étudiant profiterait sans doute d'un outil en mesure d'identifier automatiquement et de lui montrer ses utilisations inadéquates (selon une convention donnée, bien sûr) d'une construction du langage machine. Par exemple, un cadre d'appel mal restauré lors du retour d'un appel de fonction. L'identification automatique de telles situations permettrait aussi de penser à une solution qui pourrait rattacher l'erreur d'utilisation à un concept de programmation mal compris par l'étudiant, et ainsi le diriger vers des références appropriées.

ANNEXE A

SPÉCIFICATION DES INSTRUCTIONS DU SIMULATEUR DE PROCESSEUR PEP/8

Aide-mémoire Pep/8

Jean Privat

INF2170 Organisation des ordinateurs et assembleur

39 instructions de PEP 8

Spécif. instr.	Instruct.	Significations	Modes d'adressage	Codes conditions affectés
00000000	STOP	Arrête l'exécution du programme		
00000001	RETTR	Retour d'interruption		
00000010	MOVSPA	Placer SP dans A		
00000011	MOVFLGA	Placer NZVC dans A		
0000010a	BR	Branchement inconditionnel	i,x	
0000011a	BRLE	Branchement si inférieur ou égal	i,x	
0000100a	BRLT	Branchement si inférieur	i,x	
0000101a	BREQ	Branchement si égal	i,x	
0000110a	BRNE	Branchement si non égal	i,x	
0000111a	BRGE	Branchement si supérieur ou égal	i,x	
0001000a	BRGT	Branchement si supérieur	i,x	
0001001a	BRV	Branchement si débordement	i,x	
0001010a	BRC	Branchement si retenue	i,x	
0001011a	CALL	Appel de sous-programme	i,x	
0001100r	NOTr	NON bit-à-bit du registre		NZ
0001101r	NEGr	Opposé du registre		NZV
0001110r	ASLr	Décalage arithmétique à gauche du registre		NZVC
0001111r	ASRr	Décalage arithmétique à droite du registre		NZC
0010000r	ROLr	Décalage cyclique à gauche du registre		C
0010001r	RORr	Décalage cyclique à droite du registre		C
001001nn	NOPn	Interruption unaire pas d'opération		
00101aaa	NOP	Interruption non unaire pas d'opération	i	
00110aaa	DECI	Interruption d'entrée décimale	d,n,s,sf,x,sx,sxf	NZV
00111aaa	DECO	Interruption de sortie décimale	i,d,n,s,sf,x,sx,sxf	
01000aaa	STRO	Interruption de sortie de chaîne	d,n,sf	
01001aaa	CHARI	Lecture caractère	d,n,s,sf,x,sx,sxf	
01010aaa	CHARO	Sortie caractère	i,d,n,s,sf,x,sx,sxf	
01011nnn	RETn	Retour d'un appel avec n octets locaux		
01100aaa	ADDSP	Addition au pointeur de pile (SP)	i,d,n,s,sf,x,sx,sxf	NZVC
01101aaa	SUBSP	Soustraction au pointeur de pile (SP)	i,d,n,s,sf,x,sx,sxf	NZVC
0111raaa	ADDr	Addition au registre	i,d,n,s,sf,x,sx,sxf	NZVC
1000raaa	SUBr	Soustraction au registre	i,d,n,s,sf,x,sx,sxf	NZVC
1001raaa	ANDr	ET bit-à-bit du registre	i,d,n,s,sf,x,sx,sxf	NZ
1010raaa	ORr	OU bit-à-bit du registre	i,d,n,s,sf,x,sx,sxf	NZ
1011raaa	CPr	Comparer registre	i,d,n,s,sf,x,sx,sxf	NZVC
1100raaa	LDr	Placer 1 mot dans registre	i,d,n,s,sf,x,sx,sxf	NZ
1101raaa	LDBYTEr	Placer octet dans registre (0-7)	i,d,n,s,sf,x,sx,sxf	NZ
1110raaa	STr	Ranger registre dans 1 mot	d,n,s,sf,x,sx,sxf	
1111raaa	STBYTEr	Ranger registre (0-7) dans 1 octet	d,n,s,sf,x,sx,sxf	

8 directives de PEP 8

Directive	Signification
.BYTE	Réserve 1 octet mémoire avec valeur initiale.
.WORD	Réserve 1 mot mémoire avec valeur initiale.
.BLOCK	Réserve un nombre d'octets mis à zéro.
.ASCII	Réserve l'espace mémoire pour une chaîne de caractères (ex : "Chaîne").
.ADDRSS	Réserve 1 mot mémoire pour un pointeur.
.EQUATE	Attribue une valeur à une étiquette.
.END	Directive obligatoire de fin d'assemblage qui doit être à la fin du code.
.BURN	Le programme se terminera à l'adresse spécifiée par l'opérande. Ce qui suit .BURN est écrit en ROM.

Codes ASCII importants (hexadécimaux)

Code ASCII	Caractère
00	Caractère NUL
0A	Caractère de saut de ligne sur PEP 8 (Enter)
20	Espace ' '
30	Premier chiffre '0'
41	Premier caractère alphabétique majuscule 'A'
61	Premier caractère alphabétique minuscule 'a'

Adressages

Mode	aaa	a	Lettres	Opérande
Immédiat	000	0	i	Spec
Direct	001		d	mem[Spec]
Indirect	010		n	mem[mem[Spec]]
Sur la pile	011		s	mem[PP+Spec]]
Indirect sur la pile	100		sf	mem[mem[PP+Spec]
Indexé	101	1	x	mem[Spec + X]
Indexé sur la pile	110		sx	mem[PP+Spec+X]]
Indirect indexé sur la pile	111		sxf	mem[mem[PP+Spec]+X]

Registres

Symbole	r	Description	Taille
N		Négatif	1 bit
Z		Nul (Zero)	1 bit
V		Débordement (Overflow)	1 bit
C		Retenue (Carry)	1 bit
A	0	Accumulateur	2 octets (un mot)
X	1	Registre d'index	2 octets (un mot)
PP		Pointeur de pile (SP)	2 octets (un mot)
CO		Compteur ordinal (PC)	2 octets (un mot)
		Spécificateur d'instruction	1 octet
Spec		Spécificateur d'opérande	2 octets (un mot)

RÉFÉRENCES

Allen, F. E. (1970). Control Flow Analysis. *Proceedings of ACM Symposium on Compiler Optimization*, 1–19.

Avgerinos, T., Cha, S. K., Rebert, A., Schwartz, E. J., Woo, M. et Brumley, D. (2014a). Automatic exploit generation. *Communications of the ACM*, *57*(2), 74–84.

Avgerinos, T., Rebert, A., Cha, S. K. et Brumley, D. (2014b). Enhancing symbolic execution with veritesting. *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, 1083–1094.

Balakrishnan, G. et Reps, T. (2004). Analyzing Memory Accesses in x86 Executables. *Springer*, 5–23.

Balakrishnan, G., Reps, T., Melski, D. et Teitelbaum, T. (2008). WYSINWYX : What you see is not what you execute. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 4171 LNCS 202–213.

Barnett, M. et Leino, K. R. M. (2005). Weakest-precondition of unstructured programs. *The 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering - PASTE '05*, p. 82.

Beck, K. (2003). *Test-driven development : by example*. Addison-Wesley.

Beyer, D., Henzinger, T. a. et Théoduloz, G. (2007). Configurable Software Verification : Concretizing the Convergence of Model Checking and Program Analysis. *Computer Aided Verification*, *4590*, 504–518.

Bloem, R., Koenighofer, R., Rock, F. et Tautschnig, M. (2014). Automating test-suite augmentation. *Proceedings - International Conference on Quality Software*, 67–72.

Bounimova, E., Godefroid, P. et Molnar, D. (2013). Billions and billions of constraints : Whitebox fuzz testing in production. *Proceedings - International Conference on Software Engineering*, 122–131.

Cadar, C., Dunbar, D. et Engler, D. R. (2008). KLEE : Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, 209–224.

Cha, S. K., Avgerinos, T., Rebert, A. et Brumley, D. (2012). Unleashing Mayhem on binary code. Dans *Proceedings - IEEE Symposium on Security and Privacy*, 380–394. IEEE.

Cifuentes, C. et Van Emmerik, M. (1999). Recovery of jump table case statements from binary code. *Proceedings - 7th International Workshop on Program Comprehension, IWPC 1999*, 192–199.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N. et Zadeck, F. K. (1989). An efficient method of computing static single assignment form. *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '89*, 25–35.

Dillig, I., Dillig, T. et Aiken, A. (2008). Sound, complete and scalable path-sensitive analysis. *ACM SIGPLAN Notices*, *43*(6), 270.

Dimjaševic, M. (2013). JPF-Doop : Combining Concolic and Random Testing for java. *Java pathfinder workshop*. Récupéré de `https://dimjasevic.net/marko/wp-content/uploads/2013/10/jpf-workshop-2013.pdf`

Flanagan, C. et Saxe, J. B. (2001). Avoiding exponential explosion. *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '01*, 193–205.

Gargantini, A. et Heitmeyer, C. (1999). Using model checking to generate tests from requirements specifications. *ACM SIGSOFT Software Engineering Notes*, *24*(6), 146–162.

Jager, I. et Brumley, D. (2010). Efficient Directionless Weakest Preconditions. *CyLab*, p. 27. Récupéré de `http://repository.cmu.edu/cgi/viewcontent.cgi?article=1024{&}context=cylab{%}5Cnpapers2://publication/uuid/6D10F6BE-9A6C-4C63-9C5D-48BBC33AABD1`

Kinder, J. et Veith, H. (2008). Jakstab : A Static Analysis Platform for Binaries. *Computer Aided Verification*, *5123*, 423–427.

Kruegel, C., Robertson, W., Valeur, F. et Vigna, G. (2004). Static disassembly of obfuscated binaries. *USENIX Security Symposium*, p. 18.

Kuznetsov, V., Kinder, J., Bucur, S. et Candea, G. (2012). Efficient state merging in symbolic execution. *ACM SIGPLAN Notices*, *47*(6), 193–204.

Köhler, W. et Guillaume, P. (1928). L'intelligence des singes supérieurs. *Revue de Métaphysique et de Morale*, *35*(3), 5–6.

Leino, K. R. M. (2005). Efficient weakest preconditions. *Information Processing Letters*, *93*(6), 281–288.

Li, G., Ghosh, I. et Rajan, S. P. (2011). KLOVER : A symbolic execution and automatic test generation tool for C++ programs. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *6806 LNCS*, 609–615. Récupéré de `http://llbmc.org/files/papers/ASE13.pdf`

Offutt, J., Liu, S., Abdurazik, A. et Ammann, P. (2003). Generating test data from state-based specifications. *Software Testing Verification and Reliability*, *13*(1), 25–53.

Pacheco, C. et Ernst, M. D. (2007). Randoop : Feedback-Directed Random Testing for Java. *Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion - OOPSLA '07*, *5*, 815.

Philipps, J., Pretschner, A., Slotosch, O., Aiglstorfer, E., Kriebel, S. et Scholl, K. (2003). Model-based Test Case Generation For Smart Cards. *Elsevier Electronic Notes in Theoretical Computer Science*, *80*, 170–184.

Rayadurgam, S. et Heimdahl, M. (2001). Coverage based test-case generation using model checkers.

Rice, H. G. (1953). Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, *74*(2), 358–358. Récupéré de `http://www.ams.org/jourcgi/jour-getitem?pii=S0002-9947-1953-0053041-6`

Robinson, B., Ernst, M. D., Perkins, J. H., Augustine, V. et Li, N. (2011). Scaling up automated test generation : Automatically generating maintainable regression unit tests for programs. *2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011, Proceedings*, 23–32.

Schwarz, B., Debray, S. et Andrews, G. (2002). Disassembly of executable code revisited. Dans *Proceedings - Working Conference on Reverse Engineering, WCRE*, volume 2002-Janua, 45–54.

Shellphish (2018). angr Github Repository. Récupéré de `https://github.com/angr/angr`

Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C. et Vigna, G. (2016). SOK : (State of) the Art of War : Offensive Techniques in Binary Analysis. Dans *Proceedings - 2016 IEEE Symposium on Security and Privacy, SP 2016*, 138–157. IEEE.

Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M. G., Liang, Z., Newsome, J., Poosankam, P. et Saxena, P. (2008). BitBlaze : A new approach to computer security via binary analysis. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5352 LNCS 1–25. Springer, Berlin, Heidelberg.

Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C. et Vigna, G. (2016). Driller : Augmenting Fuzzing Through Selective Symbolic Execution. *Proceedings 2016 Network and Distributed System Security Symposium*.

Su, Y., Li, M., Tang, C. et Shen, R. (2016). Mitigating path explosion in symbolic execution via branches merging. *International Journal of Applied Engineering Research*, *11*(17), 9159–9165. Récupéré de `http://www.ripublication.com/ijaer16/ijaerv11n17{_}06.pdf`

Warford, J. S. et Okelberry, R. (2007). A Programmable Simulator for a Central Processing Unit The assembly and ISA levels. *SIGCSE*.

Xu, Z. et Rothermel, G. (2009). Directed test suite augmentation. *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, 406–413.