UNIVERSITE BRETAGNE LOIRE / MATHSTIC

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

THESE DE DOCTORAT DE

l'ÉCOLE NATIONALE SUPÉRIEURE MINES-TÉLÉCOM ATLANTIQUE BRETAGNE-PAYS DE LA LOIRE – IMT ATLANTIQUE

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par

Alexandre GONZALVEZ

Affiner la déobfuscation symbolique et concrète de programmes protégés par des prédicats opaques

Thèse présentée et soutenue à Rennes, le 2 Juin 2020
Unité de recherche : Lab-STICC, Irisa
Thèse N° 2020IMTA0187

Rapporteurs avant soutenance :

Pascal LAFOURCADE	Maître de conférence, HdR, Université Clermont Auvergne, Clermont-Ferrand
Daniel LE BERRE	Professeur des universités, Université d'Artois, Lens

Composition du Jury :

Président :	Jean-Louis LANET	Professeur des universités, Inria, Rennes
Examinateur :	Sébastien BARDIN	Chargé de recherche, CEA, LIST, Saclay
Rapporteur :	Pascal LAFOURCADE	Maître de conférence, HdR, Université Clermont Auvergne, Clermont-Ferrand
Rapporteur :	Daniel LE BERRE	Professeur des universités, Université d'Artois, Lens
Dir. de thèse :	Caroline FONTAINE	Directrice de recherche, CNRS, Saclay
Co-dir. de thèse :	Fabien DAGNAT	Maître de conférences, HdR, IMT Atlantique, Brest

Invité(s) :

Sébastien JOSSE	Chercheur, Direction générale de l'armement, Rennes
Marion VIDEAU	Responsable scientifique, Quarkslab, Rennes

Those who cannot remember
the past are condemned to repeat it.
George Santayana, *The Life of Reason*

Les détails font la perfection, et la perfection n'est pas un détail.
Léonard de Vinci

Préambule

Cette thèse a été financée sur projet du *Pôle d'Excellence Cyber* (PEC), par l'*École nationale supérieure Mines-Télécom Atlantique Bretagne-Pays de la Loire* (IMT Atlantique) et par l'*Institut National de Recherche en Informatique et en Automatique* (Inria). Cette thèse s'est déroulée au sein du département *Image et Traitement de l'Information* (ITI) à IMT Atlantique, au sein de l'équipe *Sécurité, Fiabilité et Intégrité de l'Information des Systèmes* (SFIIS), ainsi que dans l'équipe-projet *Threat Analysis and Mitigation for Information Security* (TAMIS) de Inria Rennes-Bretagne Atlantique (Inria-RBA) et de l'*Institut de Recherche en Informatique et Systèmes Aléatoires* (IRISA). Les adresses des différentes entités sont données ci-dessous :

IMT Atlantique Bretagne-Pays de la Loire
Campus de Brest
Technopôle Brest-Iroise
CS 83818
29238 Brest cedex 03
France

Inria Rennes - Bretagne Atlantique
Campus universitaire de Beaulieu
263 Avenue du Général Leclerc
35042 Rennes cedex
France

IRISA Rennes
Institut de Recherche en Informatique et Systèmes Aléatoires
Campus universitaire de Beaulieu
263 Avenue du Général Leclerc
CS 74205
35042 Rennes cedex
France

Pôle d'Excellence Cyber
12 rue Patis Tatelin
35000 Rennes
France

Remerciements

Merci, merci, merci : une étape se termine et une autre commence. C'est avec ce simple concept de remerciement que je voudrais exprimer ma gratitude envers tous ceux qui ont participé directement ou indirectement au succès de cette thèse.

Merci infiniment à ma directrice de thèse Caroline Fontaine, et à mon directeur de thèse (arrivé plus tardivement) Fabien Dagnat, d'avoir su m'aiguiller et me soutenir tout du long de ce travail de thèse. Merci à Sébastien Josse, encadrant de cette thèse, pour le support et les conseils avisés sur le vaste sujet de l'obfuscation, malgré son emploi du temps chargé. Merci aux différents membres du jury d'avoir évalué mes travaux : à Jean-Louis Lanet de m'avoir fait l'honneur de présider mon jury, et pour son soutient morale à un ancien de Limoges, aux rapporteurs Pascal Lafourcade et Daniel Le Berre pour leurs remarques constructives qui ont permis d'améliorer ce manuscrit, à Sébastien Bardin pour ses observations pertinentes, et à Marion Videau pour ses commentaires avisés. Ces travaux n'auraient pas été possibles sans le soutien du PEC, de IMT-Atlantique et de Inria : je remercie ces différentes entités ainsi que leurs personnels administratifs.

Une thèse s'effectue au sein d'une équipe de recherche. J'ai eu la chance d'être dans deux équipes situées aux abords de deux villes en Bretagne : Brest et Rennes. J'ai eu l'occasion de croiser et de discuter avec un grand nombre de personnes pour une idée, une actualité, un café ou un gâteau. Vous êtes nombreux à avoir eu un effet positif et je ne pourrais pas tous vous citer ici. Je tiens quand même à remercier plus particulièrement ceux qui ont dû me supporter au quotidien. Merci à Olivier Zendra de m'avoir soutenu et permis de réaliser cette thèse dans son équipe. Merci à mes co-bureaux (Ronan, Bruno, Cassius) pour leurs échanges avisés, leurs aides et le café. Merci à la maman de Cogito, Alix, pour l'acclimatation les premières semaines. Merci à Nisrine pour la *positive attitude* et les différents voyages. Merci à Delphine pour sa bonne humeur et son écoute. Merci à Tania pour ces (longues) discussions et échanges sur des sujets avancées et à l'accès à ses nombreux livres. Merci à Kévin, pour des voyages improbables

et des clichés magnifiques. Merci aux anciens doctorants (Aurélien, Mounir, Tristan, Edwin, Guillaume) et à ceux en devenir (Bonne chance !) (Christophe, Lamine, Duy, Cassius, Routa, Farah, Léopold, Vasile, Mathieu, Benoît, Cédric) de m'avoir aidé à connaître le chemin. Merci plus particulièrement à Olivier Decourbe qui m'a particulièrement aidé dans cette aventure, mais qui n'a pas eu la chance de pouvoir la finir comme prévu. Merci à vous aussi : Annelie, Ioana-Domnina, Greg, Benji, Louis, Pierre-Yves, Laurent, Najah, Céline, Yoann, Nicolas, Ludovic, Yulliwas, Florian, Colas, Martin, Pierrick, Jeffrey, Julien, Agathe, Matthieu ... et à ceux que j'oublie (pardon).

Le temps passe, beaucoup de choses ont changé, mais qui aurait pu imaginer que le temps se serait si vite écoulé. Merci à vous aussi les amis, au serveur Jupiter, et à la famille pour vos soutiens durant les tempêtes traversées. Enfin, j'aimerais remercier toutes les personnes croisées avant cette thèse qui ont permis de rendre ce moment faisable, sans forcément le savoir : merci à vous qui m'avez apporté une vision différente du monde. *Alea jacta est*

Table des matières

1 Introduction **1**

1.1 Contexte 1
1.2 Questions de recherche 2
1.3 Nos résultats 4
1.3.1 Défi 1 : relation entre langage machine et opacité 4
1.3.2 Défi 2 : amélioration de la détection des prédicats opaques 5
1.3.3 Défi 3 : propositions pour réduire l'opacité 5
1.3.4 Défi 4 : aider les études futures 6
1.4 Plan de lecture 6

I Les prédicats opaques et la déobfuscation **9**

2 Prédicats opaques utilisés en obfuscation **11**

2.1 Introduction 11
2.1.1 L'obfuscation 11
2.1.2 Un peu d'histoire 18
2.1.3 L'obfuscation du graphe de flot de contrôle 19
2.2 Modèles de sécurité et d'attaquant 22
2.2.1 Modèles de sécurité 22
2.2.2 Modèles d'attaquant 24
2.3 Les prédicats opaques 25
2.3.1 Définitions et limites 26
2.3.2 Méthodes de détections classiques 32
2.3.3 Autres travaux associés 34
2.4 Conclusion du chapitre 35

3 Déobfuscation symbolique et concrète **37**

3.1 La déobfuscation automatique 37
3.1.1 L'analyse statique automatisée 38

3.1.2 Exemple de déobfuscation 41
3.2 Exécution dynamique symbolique 43
3.2.1 Exécution symbolique et exécution concrète 44
3.2.2 Deux approches différentes 50
3.2.3 Exemples de moteurs d'exécution symbolique dynamique 53
3.3 Mécanismes des solveurs *SMT* 58
3.3.1 Du solveur SAT au solveur *SMT* 59
3.3.2 Raisonnement automatique à l'aide d'une théorie $\mathcal{T}$ 63
3.4 Exemple d'un effet observable lors de l'analyse d'un prédicat opaque 72
3.5 Conclusion du chapitre 74

II Améliorations de l'analyse des prédicats opaques 75

4 Un équilibre difficile entre protection logicielle et intégrité de programme 77
4.1 Introduction 77
4.2 Présentation de la machine 79
4.2.1 Architecture de la machine 79
4.2.2 Sémantiques utilisées 81
4.2.3 Description des *ISAs* 81
4.3 Protection et intégrité du flot de contrôle 81
4.3.1 Protection du flot de contrôle à l'aide de prédicat opaque à trappes . . . 83
4.3.2 Intégrité du flot de contrôle (*CFI*) 83
4.4 Recouvrement du *CFG* 85
4.5 Tests et améliorations 86
4.5.1 Exemple avec l'*AES* 90
4.5.2 Résultats et observations 91
4.5.3 Recouvrement des distributeurs 93
4.6 Discussions 96
4.6.1 Limites de la *CFI* 96
4.6.2 Programmes acceptés et rejetés 99
4.7 Conclusion du chapitre 101

5 Notre approche sur la détection de prédicats opaques 103
5.1 Introduction 103

5.1.1 Problème . . . 104
5.1.2 Objectifs et contributions . . . 104
5.1.3 Travaux associés . . . 106
5.2 Définitions et propriétés . . . 106
5.2.1 Structure . . . 107
5.2.2 Le problème d'isomorphisme et d'équivalence . . . 110
5.2.3 Structure calculable, décidable ou automatique . . . 110
5.2.4 Présentation automatique et résolution d'une requête . . . 112
5.2.5 Croissance de recouvrement . . . 117
5.3 Détection de structure indépendante . . . 118
5.3.1 Conjecture et justifications . . . 118
5.3.2 Mise en place d'une mesure de détection . . . 119
5.3.3 Intégration dans KLEE . . . 121
5.3.4 Exemple détaillé . . . 121
5.4 Évaluation et Expériences . . . 123
5.4.1 Évaluation . . . 123
5.4.2 Configuration et Jeux de données . . . 124
5.4.3 Résultats et observations . . . 125
5.5 Discussions . . . 126
5.5.1 Méthodologie . . . 126
5.5.2 Limites de cette méthode et pistes futures . . . 126
5.6 Conclusion du chapitre . . . 127

6 Notre proposition pour réduire l'opacité d'un prédicat opaque 129
6.1 Introduction . . . 130
6.1.1 Problème . . . 130
6.1.2 Objectifs et contributions . . . 131
6.1.3 Exemple type . . . 132
6.2 Prédicat et représentations . . . 136
6.2.1 Prédicat et solutions . . . 136
6.2.2 Vérité et stabilité de l'interprétation . . . 137
6.3 Redéfinition d'une expression *SMT* . . . 138
6.3.1 Décomposition d'une expression *SMT* . . . 139
6.3.2 Méthode . . . 139

6.3.3 Implémentation 141
6.4 Évaluations et expériences 141
6.5 Travaux existants 143
6.6 Conclusion du chapitre 144

7 Participation à la compétition SMT-COMP 145
7.1 Introduction 145
7.2 Requêtes *SMT* 146
7.2.1 Description 146
7.2.2 Description de la construction du premier type de requêtes 147
7.3 Résultats de la compétition et interprétations 152
7.4 Conclusion du chapitre 155

III Conclusion générale 157

8 Conclusion générale 159
8.1 Résumé des contributions 159
8.2 Perspectives 160
8.2.1 Caractérisation des prédicats opaques en fonction d'une théorie 160
8.2.2 Extraction automatique de prédicats opaques 161
8.2.3 Réalisation d'un générateur de prédicats opaques 161

Bibliographie 163

Diffusion des résultats 181

TABLE DES FIGURES

2.1 Paradigme de simulation 15
2.2 Chronologie simplifiée sur les évolutions techniques de l'obfuscation 20
2.3 Obfuscation du graphe de flot de contrôle avec un « dispatcheur » central 21
2.4 Obfuscation du graphe de flot de contrôle avec remplacement des sauts directs . 22
2.5 Modèle de sécurité par simulation (*VBB*) 23
2.6 Modèle de sécurité basé sur l'indistinguabilité (*iO*) 24
2.7 Fonctionnement d'un prédicat opaque statique 30
2.8 Fonctionnement d'un prédicat opaque dynamique 30
2.9 Fonctionnement d'un prédicat opaque aléatoire 31

3.1 Représentation d'un programme 38
3.2 Tableau T en mémoire (lecture seule) 42
3.3 Exemple de déobfuscation basée sur [Yad16] 42
3.4 Programme exemple TOY 44
3.5 Exemple de représentation *imparfaite* (la relation $\oplus$ n'est pas conservée) 45
3.6 Magasin symbolique du programme TOY 46
3.7 Condition du programme TOY 46
3.8 Modèles concrets de la condition du programme TOY 47
3.9 Nouveau magasin symbolique du programme TOY 49
3.10 Nouvelle condition du programme TOY 49
3.11 Fonctionnement simplifié de `KLEE` 56
3.12 Pile de solveur de `KLEE` 56
3.13 Exemple d'état symbolique dans `KLEE` 57
3.14 Fonctionnement simplifié de `Angr` 58
3.15 Approche *Eager* 65
3.16 Approche *Lazy* 67
3.17 Stratégie de l'utilisation de la théorie QF-AUFBV 73

5.1 Mesures temporelles aux interfaces (points marron) : cas pratique avec le solveur `Z3` 122

6.1 Avec `KLEE-Z3`, mesures du taux de croissance temporel des appels au SAT-solveur pour la fonction `APhash` (32-bits), pour des entrées de 16, 32, 48, 64 lettres 143

Liste des tableaux

1.1 Comparaison à haut niveau du chiffrement de données et de l'obfuscation de programme, d'après les travaux de [HB15] 2

2.1 Exemples de prédicats opaques arithmétiques 29

3.1 Exemples de prédicats de chemin, de symbolisation et de concrétisation, du programme exemple TOY 48
3.2 Règles de transitions pour CDCL 62
3.3 Résolution de la formule F 63
3.4 Règles de transitions supplémentaires pour CDCL($\mathcal{T}$) 68

4.1 Description des instructions 82
4.2 Ensemble d'instructions pour un AES pour chaque ISA. 90
4.3 Évaluation des performances des ISAs et du recouvrement du CFG pour une implémentation de l'AES 92
4.4 Nombre d'instructions par répartiteur (pour $p > 3$) 94
4.5 Latence de répartiteur 95

5.1 Taux de croissance de recouvrement maximal 123
5.2 Quantité de prédicats opaques détectés 125

6.1 Règles de redéfinition des opérations 140
6.2 Mesures approximatives de la détection de prédicats opaques et des temps d'analyse de programmes contenant des prédicats opaques, avant et après la redéfinition des requêtes *SMT*. (Temps en secondes) 143

7.1 Description courte des requêtes *SMT* 148
7.2 Comportements observés 153
7.3 Résultats observés 153
7.4 Temps CPU (en secondes) 154
7.5 Quantité de mémoire utilisée (en méga-octets) 154

Liste des Algorithmes

1 Essais concoliques [Cha+19] 51
2 Essais générés par exécutions [Cha+19] 54
3 Un pseudo-code des solveurs SAT modernes [PD10] 60
4 Algorithme CDCL($\mathcal{T}$) [CS18] 69

5 Règles pour l'exécution d'un programme 80
6 Algorithme d'extraction du CFG structurel 87
7 Suivi de l'algorithme d'extraction du CFG, utilise l' algorithme 8 pour la fonction de hachage `ReturnStack` 88
8 Algorithme de hachage `ReturnStack` 89

9 Détection des Prédicats opaques 120

LISTINGS

3.1 Code statique 42
3.2 Premier modèle 42
3.5 Exemple de programme 52
3.6 Exemple de prédicat opaque dans un programme C 72
4.1 Masquer le flot de contrôle avec un prédicat à trappe 83
4.2 Implémentation assembleur de Xtimes (sous fonction de l'AES) avec l'ISAv1 . 90
4.3 Motif du répartiteur ISAv1 93
4.4 Motif du répartiteur ISAv2 93
4.5 Motif du répartiteur ISAv3 94
4.6 Illustration d'un branchement conditionnel à n-sorties 96
4.7 Machine virtuelle subleq 96
5.1 Exemple de prédicat opaque dans un programme C 121
5.2 Exemple de fonction insérée dans un programme par *Tigress* pour fabriquer un prédicat opaque (représentation équivalente en code C à partir d'un binaire) . . 124
6.1 Exemple type 133
6.2 Fonction de hachage `APhash` [Par13] « `Copyright 2020 Arash Partow` » `General Hash Function License` selon `MIT License` (`https://www.partow.net/programming/hashfunctions/#GeneralHashFunctionLicense`) . . . 134
6.3 Début de la requête *SMT* fabriquée par `angr` 135
7.1 Requête *SMT* : `opStructure_NPT_1.smt2` 149
7.2 Fonctions principales utilisées pour fabriquer des prédicats opaques 151

Chapitre 1

Introduction

1.1 Contexte

De nombreux ordinateurs et appareils sont aujourd'hui connectés à Internet. Leurs logiciels apparaissent comme des cibles privilégiées pour des pirates ou des développeurs de *programmes malveillants* (*malwares*). Autrement dit, des personnes peuvent chercher à modifier, voler, ou supprimer les informations contenues dans un logiciel, en utilisant des techniques de *rétro-ingénierie* (*reverse engineering*). Pour se protéger de ces attaques, les développeurs de logiciels ont mis en place des protections logicielles. Parmi les différentes solutions de protections logicielles existent les techniques d'*obfuscation* [1]. L'obfuscation regroupe l'ensemble des techniques qui allongent le temps de compréhension d'un objet, dont l'objectif est de rendre facilement compréhensible l'objet pour un ensemble de personnes, et difficile sinon. L'objectif de l'*obfuscation logicielle* (*software obfuscation*) est d'étendre le temps nécessaire pour comprendre des informations et des algorithmes critiques contenus dans des logiciels publiés. En pratique, le logiciel publié doit conserver la fonctionnalité et la même performance temporelle (ou dans certains cas avec un léger ralentissement) du logiciel original (ou du code binaire original). Cohen [Coh93] présente le premier travail académique sur la transformation de programme qui met en place des protections par ajout de confusion. Depuis, nous pouvons compter deux chemins de recherche académique. Le plus ancien chemin débuta avec les travaux de Collberg [CTL97 ; CTL98] qui définit le concept d'obfuscation pour la première fois . Depuis, de nombreux travaux ont proposé des techniques utilisables pour des programmes réels [NC09 ; HD11 ; Sch+16]. Les développeurs de programmes malveillants ont parfois repris ces techniques pour protéger leur charge virale [YY10]. Le second chemin commença avec les travaux académiques de Barak [Bar+01]. Cette nouvelle voie permet d'aborder l'*obfuscation* par un formalisme plus algébrique. Une apparente analogie avec la cryptographie existe, comme le montre le tableau 1.1. De nombreux travaux de la communauté cryptographique explorent ce chemin, dans le but de présenter une

1. L'usage du terme obfuscation en français est un néologisme d'emprunt lexical tiré de l'anglais *obfuscation* venant du latin *obfuscare* (construit avec *ob* : devant, *fuscus* : sombre).

nouvelle génération d'algorithmes cryptographiques [HB15]. Malheureusement, au moment de la rédaction de ce manuscrit, les résultats de cette voie sont difficilement employables dans des logiciels réels.

	Chiffrement des données	**Obfuscation de programmes**
Représentation	Éléments de groupes, d'anneaux ...	Circuits, Machines de Turing ...
Modélisation de la sécurité	*Sécurité sémantique* : soit M_0 M_1, ($\|M_0\|$=$\|M_1\|$) $Enc(M_0) \sim Enc(M_1)$	*Obfuscation indistinguable* (iO) : soit $P_1 = P_2$, ($\|P_1\| = \|P_2\|$) $Obf_{iO}(P_1) \sim Obf_{iO}(P_2)$
Méthode pour cacher l'information	Randomisation	Randomisation
Méthode pour retrouver l'information	Utilisation d'une clé secrète	Exécution de tous les états *légaux* du programme

TABLE 1.1 – Comparaison à haut niveau du chiffrement de données et de l'obfuscation de programme, d'après les travaux de [HB15]

Parmi les différentes techniques d'*obfuscation logicielle*, l'utilisation de *prédicats opaques* (*opaque predicate*) est une méthode simple et efficace. Un prédicat est une expression conditionnelle pouvant être évaluée à vrai ou faux. Un prédicat est opaque lorsqu'il est toujours évalué à la même valeur de vérité, et est difficile à comprendre par un adversaire. Plus particulièrement, ce sont des prédicats qui contiennent un ensemble d'opérations redéfinies dans le but de gêner une analyse de *rétro-ingénierie* [Bio+17]. Les premiers travaux de COLLBERG [CTL97 ; CTL98] proposèrent quelques *prédicats opaques*. Leurs faibles coûts et leur caractère furtif en font des objets facilement utilisables dans des programmes. De ce fait, le *graphe de flot de contrôle* [2] (*control flow graph* (*CFG*)) d'un programme peut contenir des chemins infaisables et la complexité de compréhension du programme augmente [Anc+07]. L'analyse d'un programme contenant des *prédicats opaques* devient plus difficile.

1.2 Questions de recherche

L'analyse automatique d'un programme fait appel à des méthodes de *rétro-ingénierie*. Les techniques d'*obfuscation* permettent de se prémunir contre cette *rétro-ingénierie*. En conséquence, lorsqu'une méthode d'obfuscation protège un programme, une analyse automatique a besoin de réduire l'effet de la technique d'*obfuscation* pour réussir son analyse. Cette étape est appelée la *déobfuscation*. Plus cette opération de *déobfuscation* ajoute du temps lors d'une analyse automatique, plus nous pouvons dire que la technique d'*obfuscation* est robuste pour cette analyse. Malheureusement, à l'heure actuelle, aucune modélisation générale ne fait consensus

2. Le graphe de flot de contrôle est une représentation sous forme de graphe, du flot d'exécution d'un programme.

pour comprendre pourquoi une technique d'*obfuscation* est plus ou moins robuste. Seule une étude de cas précise permet de répondre à cette interrogation. Aussi, la robustesse d'une méthode d'*obfuscation* particulière ne sait aujourd'hui être évaluée que par rapport à une famille d'outils d'analyse particulière. Pour étudier des *prédicats opaques* , nous avons choisi d'utiliser des outils d'analyses symboliques et concrètes, et plus particulièrement les moteurs d'*exécution symbolique dynamique* (*dynamic symbolic execution* (*DSE*)). L'avantage de ces moteurs est qu'ils permettent d'obtenir des analyses rapides et précises de programmes avec très peu de faux positifs. Ces moteurs s'appuient sur des solveurs de contraintes décrites dans une théorie du premier ordre, appelés solveurs de *satisfiabilité modulo theorie (SMT)*. Comme la sous-section 2.3.1 le présente, dans un programme obfusqué, les *prédicats opaques* remplacent les prédicats des conditions de branchement, ce qui participe à les rendre difficilement détectables. De plus, ces prédicats n'ont pas pour but d'influencer l'exécution du programme. Plusieurs familles de *prédicats opaques* existent, mais les membres de ces familles sont rarement utilisés seuls. Les *prédicats opaques* ne sont pas prévus pour être optimisés lors de la compilation du programme [Sch+16]. Les connaissances actuelles sur les *prédicats opaques* ne permettent pas de dire si les méthodes de détections ou d'inhibitions sont suffisantes pour toutes les familles de *prédicats opaques*. Compte tenu de l'importance de cette difficulté, nous nous interrogeons sur les relations existantes entre les mécanismes de capture de l'information réalisés par les outils d'analyses et les *prédicats opaques*. C'est pourquoi nous soulevons la question suivante :

Peut-on améliorer la compréhension des mécanismes qui rendent les prédicats opaques efficaces en tant que technique d'obfuscation, afin d'améliorer les méthodes d'analyses symboliques et concrètes ?

Une réponse à cette question peut permettre d'améliorer les outils de *déobfuscation* qui utilisent des outils d'analyse automatique, fonctionnant sur des méthodes symboliques et concrètes. À partir de cette question, nous pouvons définir quatre défis à relever :

Défi 1 : « Comprendre comment le jeu d'instructions défini pour une architecture machine peut permettre l'apparition des prédicats opaques ? » Les *prédicats opaques* peuvent apparaître comme un ensemble de relations définies dans un *jeu d'instructions machine*, exécutables pour une architecture donnée. Notre objectif est de comprendre comment un jeu d'instructions rend possible l'utilisation de *prédicats opaques*.

Défi 2 : « Peut-on détecter efficacement les prédicats opaques ? » Si nous comprenons mieux la structure intrinsèque des *prédicats opaques*, nous pourrons proposer une nouvelle

méthode de détection adaptable aux outils que nous utilisons.

Défi 3 : « Peut-on comprendre comment les prédicats opaques résistent à un outil d'analyse automatique ? » Une réponse plus éclairée sur les mécanismes de capture de l'information et des raisonnements automatiques peut nous aider à affiner l'analyse des *prédicats opaques*.

Défi 4 : « Peut-on proposer des requêtes SMT contenant des prédicats opaques ? » Pour faciliter les futurs études des *prédicats opaques*, nous pourrions mettre à disposition quelques requêtes *SMT* contenant des *prédicats opaques*.

1.3 Nos résultats

Affiner la *déobfuscation* de programmes protégés par des *prédicats opaques* nécessite de bien comprendre quels sont les modèles calculables des *prédicats opaques*. Nous limitons notre travail à l'interprétation des *prédicats opaques* par des outils existants et disponibles. Nos contributions à ce travail sont les suivantes. Notre première contribution porte sur le rôle du *jeu d'instructions machine* dans la protection d'un programme, et nous permet de répondre à notre premier défi. La deuxième contribution explore les mécanismes automatiques qui permettent de capturer l'information contenue dans un programme, afin de mieux détecter les *prédicats opaques* et de proposer une réponse à notre deuxième défi. Pour répondre à notre troisième défi, nous suggérons une méthode pour redéfinir un *prédicat opaque* dans une requête *SMT*, dans le but de simplifier l'analyse réalisée par un moteur *DSE*. Pour finir, notre quatrième défi donne de la matière pour faciliter les études futures sur les *prédicats opaques*. Nous présentons maintenant un peu plus en détail chacune de ces contributions.

1.3.1 Défi 1 : relation entre langage machine et opacité

Un programme est interprété par une machine réelle grâce à un *jeu d'instructions machine* (*Instruction Set Architecture* (*ISA*)). Si la chaîne d'instructions prévue au moment de la compilation d'un programme source ne peut être modifiée durant l'exécution, alors un programme peut être considéré comme sécurisé. Une des manières de garantir cette propriété est de mettre en place un système d'*intégrité du flot de contrôle* (*Control Flow Integrity* (*CFI*)) durant l'exécution. Ce dernier repose sur la capacité du système à pouvoir capturer automatiquement le *graphe du flot de contrôle*. Dans cette contribution, nous suggérons qu'en général les prédicats à trappe,

une sous-famille de *prédicats opaques*, ne permettent pas de réaliser cette capture automatique. Nous détaillons un moyen de surmonter cette contradiction en limitant la sémantique de l'architecture du jeu d'instructions. Nous montrons plus particulièrement que l'interdiction des *sauts indirects* (*indirect jumps*) permet d'obtenir une extraction du *graphe de flot de contrôle* précis pour tous les programmes acceptables. En conséquence, nous pouvons dire que la présence d'au moins une instruction du type *saut indirect* dans un programme permet l'apparition de *prédicats opaques*. Nous suggérons l'adoption d'un *jeu d'instruction machine* restreint afin d'obtenir des programmes sécurisés.

1.3.2 Défi 2 : amélioration de la détection des prédicats opaques

Nous définissons un programme comme une *structure automatique* (*automatic structure*), c'est-à-dire une structure qui parcourt un certain chemin depuis un état de départ vers un état d'arrivée. Les outils d'analyse statique réalisent un équilibre entre plusieurs défis, comme la précision ou le temps de l'analyse. Lorsqu'un programme contient des *prédicats opaques*, ces outils rencontrent des difficultés pour respecter leur équilibre. Pour illustrer cette situation, nous rappelons le fonctionnement de la résolution d'une requête dans une théorie interprétable par un solveur *SMT*. Nous remarquons que le temps de résolution d'une requête dépend de la structure présente dans la requête. Nous conjecturons qu'un *prédicat opaque* est interprétable comme une structure indépendante. Pour vérifier notre conjecture, nous mettons en place un détecteur de *prédicats opaques* dans le moteur *DSE* `KLEE` et différents solveurs (`Z3` et `Boolector`) et nous le testons sur des programmes qui contiennent des *prédicats opaques*. Nous observons que nous arrivons à détecter une majorité de *prédicats opaques*. Nous suggérons que le choix d'une théorie pour interpréter un *prédicat opaque* influence l'opacité de ce dernier. Comprendre l'origine de cette opacité pour une théorie donnée est une nouvelle piste intéressante.

1.3.3 Défi 3 : propositions pour réduire l'opacité

Pour résoudre un ensemble de contraintes, un moteur *DSE* utilise un solveur *SMT*. Ces contraintes sont représentées dans une requête définie dans une théorie. Dans le cas des *prédicats opaques*, nous conjecturons qu'un *prédicat opaque* peut être défini de plusieurs manières dans différentes théories décidables. Nous proposons pour une théorie fixée que lors d'une détection d'un *prédicat opaque*, nous puissions réécrire l'expression définie dans la requête *SMT*. Cette réécriture peut amener à remplacer la requête par un ensemble de sous-requêtes plus faciles à résoudre que l'originale. Pour tester notre heuristique, nous proposons de redéfinir des requêtes

SMT, puis de tester notre heuristique sur un ensemble restreint de programmes qui contiennent des *prédicats opaques*. Nous suggérons qu'un travail plus avancé permettrait de classer les différentes représentations d'un prédicat en fonction de la difficulté à l'analyser pour une théorie fixée.

1.3.4 Défi 4 : aider les études futures

Pour aider les études futures des prédicats opaques, nous construisons des requêtes *SMT* contenant des prédicats opaques, pour la compétition SMT-COMP 2019. Nous mettons en application nos connaissances pour construire ces requêtes. Nous constatons que les meilleurs solveurs *SMT* de la théorie choisie n'arrivent pas à résoudre une partie de ces requêtes dans le temps imparti. Ces requêtes sont disponibles librement dans la bibliothèque SMT-LIB. Nous suggérons qu'une étude plus formelle permettrait de mieux appréhender les résultats observés.

1.4 Plan de lecture

Dans ce manuscrit, nous présentons dans la partie I un état de l'art de l'utilisation des prédicats opaques en obfuscation. La partie II présente une explication des mécanismes structurels des *prédicats opaques*, pour adapter des outils d'analyse de programmes contenant des *prédicats opaques*. Notre objectif est de comprendre en quoi un *prédicat opaque* est opaque, et comment nous pouvons utiliser cette information pour améliorer nos outils d'analyse.

La partie I est constituée de deux chapitres. Le chapitre 2 introduit les *prédicats opaques* utilisés en *obfuscation*. Ce chapitre montre l'évolution de l'utilisation de l'*obfuscation* en informatique et les différents modèles de sécurité et d'attaquant définis à ce jour. Le chapitre 3 introduit les difficultés rencontrées lors de la *déobfuscation* automatique des *prédicats opaques*. À partir du problème de reconstruction d'un *control flow graph (CFG)*, ce chapitre présente les solutions apportées par des moteurs d'analyse symbolique et concrète qui s'appuient sur des solveurs *SMT* pour résoudre ce problème. Plusieurs analyses détaillées sont ensuite proposées pour illustrer les impacts négatifs des *prédicats opaques* sur ces outils. La partie II est constituée de quatre chapitres. Le chapitre 4 propose une analyse concrète du rôle d'un *jeu d'instruction machine* dans l'apparition des *prédicats opaques*, et du conflit existant avec l'*intégrité du flot de contrôle*. Le chapitre 5 met en avant certaines limites de la résolution d'une requête définie dans une théorie. Nous proposons une nouvelle méthode de détection des *prédicats opaques* en utilisant ces limites. Le chapitre 6 présente une extension du chapitre précédent, en propo-

sant une méthode de réécriture des requêtes *SMT*, pour faciliter la résolution des requêtes. Le chapitre 7 présente la construction de requêtes *SMT* contenant des *prédicats opaques*, soumises à la compétition SMT-COMP 2019, pour aider des études futures sur ce sujet. Le chapitre 8 conclut l'ensemble des contributions de ce manuscrit et discute des voies ouvertes par ce travail de recherche.

Première partie

Les prédicats opaques et la déobfuscation

CHAPITRE 2

PRÉDICATS OPAQUES UTILISÉS EN OBFUSCATION

Sommaire

2.1 Introduction 11
2.2 Modèles de sécurité et d'attaquant 22
2.3 Les prédicats opaques 25
2.4 Conclusion du chapitre 35

Résumé Nous présentons l'obfuscation dans la section 2.1 au travers de trois définitions. Nous discutons les avantages et les limites de chacune de ces définitions, pour montrer que l'obfuscation est un concept difficile à cerner. Nous décrivons dans la section 2.2 les modèles de sécurité et d'attaquant les plus couramment utilisés pour les méthodes d'obfuscation. Un contexte précis est prévu pour chaque modèle. Nous exposons dans la section 2.3 l'utilisation des prédicats opaques en obfuscation. Nous abordons leurs avantages et leurs faiblesses.

2.1 Introduction

2.1.1 L'obfuscation

Un programme informatique est défini comme une collection d'opérations permettant de manipuler une information. L'*obfuscation* regroupe les méthodes ayant pour but de rendre ces opérations facilement intelligibles pour une collection de destinataires, et difficiles pour tous les autres. Les méthodes d'obfuscation sont utilisées dans des applications variées, comme le montrent ces quelques exemples : utilisation de leurres pour perdre un attaquant (*decoy strategy*) [Coh06], mise en place d'une documentation excessive pour décourager un attaquant (*excessive documentation*) [Kaf07 ; BTT+16], ou pollution de profilage numérique (*polluting profiling*) [Car11]. Dans ce manuscrit, le terme obfuscation se limite à l'obfuscation logicielle

(*software obfuscation*), c'est-à-dire aux transformations de données et de codes (*data and code obfuscation*) [Sch+16]. L'obfuscation logicielle transforme un programme en un autre programme, de telle sorte que l'extraction de certaines informations internes contenues dans le programme d'origine (par exemple : son comportement ou ses transitions) devient plus difficile à interpréter après transformations. Ces transformations ne doivent pas (ou peu) affecter l'exécution du programme [Cec+17 ; Hos+18 ; Cec+19]. Cette transformation est intéressante financièrement lorsque le coût financier et temporel pour contourner les effets de l'obfuscation dépasse le gain potentiel d'extraction du contenu seul dans le programme [App02].

Il n'existe pas une méthode unique et générique pour réaliser de l'obfuscation logicielle, mais des familles de transformations [BS05 ; NC09 ; HB15 ; Sch+16 ; Xu+17]. À ce jour, la comparaison scientifique de ces transformations reste une tâche difficile [McD12 ; Yad+15 ; Sch+16 ; Cec+17 ; BP18]. L'origine de cette difficulté semble venir du fait qu'il existe plusieurs définitions de l'obfuscation, peu contraignantes, permettant un large choix de solutions. Pour limiter cette difficulté, nous souhaitons croiser trois définitions de l'obfuscation et discuter leur complémentarité. Nous discuterons plus en détail ces définitions et leurs limites dans les sections suivantes. La définition de l'obfuscation donnée par Brunton et Nissenbaum [BN15] aborde cette transformation consistant en l'addition d'un « bruit » dans un programme, par exemple l'ajout de transitions ou d'opérations « inutiles » dans un programme. Les travaux de Collberg, Thomborson et Low [CTL97] [1], et ceux qui suivirent [NC09] définissent l'obfuscation plus simplement, comme une transformation de programme. L'action du « bruit » sur le programme est caractérisée pour un ensemble de techniques au travers de différentes métriques. Les travaux plus théoriques de Barak *et al.* [Bar+01], définissent l'obfuscation d'un point de vue algébrique. L'obfuscation y est décrite comme la transformation d'un programme défini pour une *machine de Turing* (*Turing machine*) en général. L'obfuscation respecte trois propriétés : *correct* (fonctionnalité), *ralentissement polynomial*, et *boîte noire virtuelle*. Barak *et al.* montrent l'existence d'un ensemble de fonctions qui ne peuvent pas cacher toutes les informations qu'elles contiennent (par exemple : les fonctions avec des entrées auxiliaires (*obfuscation with auxiliary input*) [GK05 ; GK13 ; Bit+13]). Ce résultat fondamental a deux conséquences. La première est que la propriété de *boîte noire virtuelle* ne peut pas être respectée en général. Le deuxième réside dans l'existence d'un nombre restreint de classes de programmes pouvant respecter la propriété de *boîte noire virtuelle* dans certaines conditions . Par exemple, ces classes peuvent être : les programmes basés sur les fonctions point, les programmes utilisant certaines primitives cryptographiques, des mécanismes de contrôles d'accès ou les programmes utilisant des

1. Temporellement, ce sont les premiers travaux sur le sujet de l'obfuscation.

supports matériels [Can97 ; CD08 ; CRV10 ; BR14a ; Bar+14]. Les transformations appliquées à ces classes de programmes font toujours l'objet de travaux, car elles restent mal comprises. Dans ce même article, Barak *et al.* [Bar+01] proposent aussi une définition moins stricte de la propriété de *boîte noire virtuelle*, permettant de définir l'*obfuscation indistinguable* (*iO*). Dans ce cas, l'attaquant arrive difficilement à distinguer deux programmes obfusqués, de même taille (ou presque), et calculant la même fonction. L'attaquant ne peut observer que les couples d'entrées / sorties et connaître la fonctionnalité du programme. Ce dernier modèle a permis de nombreux résultats prometteurs [Gar+13 ; CLT13 ; GGH15 ; Bar+19], mais les applications dans la vie réelle restent encore incertaines [Apo+14 ; HJ16 ; Che+16 ; Cor+16 ; CMR17]. L'*obfuscation* est un domaine de recherche demandant encore un fort besoin d'activités pour relier et uniformiser les connaissances empiriques et théoriques.

2.1.1.1 Définitions de l'obfuscation

Afin de mieux cerner la notion d'*obfuscation*, nous en présentons trois définitions différentes. Nous souhaitons montrer que suivant le contexte de travail, la définition de l'obfuscation s'adapte à ce contexte. Nous commençons par la définition proposée par Brunton et Nissenbaum [BN15] :

Définition 2.1 [BN15] *L'*obfuscation *se modélise comme une information bruitée ajoutée à un signal (d'information) existant, dans le but de fabriquer une collection de données plus ambiguë, confuse, difficile à exploiter et donc moins utile.*

Les auteurs de cette première définition mettent l'accent sur la nécessité d'utiliser l'*obfuscation* comme un moyen de se prémunir d'un adversaire, non clairement défini ici. Si nous considérons le *signal* comme la connaissance contenue dans le programme, alors cette définition inscrit l'*obfuscation* comme un moyen de rendre plus difficile n'importe quelle technique de rétro-ingénierie.

Les travaux de Collberg, Thomborson et Low [CTL97] donnent une vision plus réaliste de l'*obfuscation*, en la définissant comme une transformation de programme, c'est-à-dire que l'*obfuscation* est une fonction acceptant en entrée un programme source et retournant en sortie un programme cible.

Définition 2.2 [CTL97] *Soit* $P \xrightarrow{T} P'$ *la transformation* T *d'un programme source* P *vers un programme cible* P'. $P \xrightarrow{T} P'$ *est une* transformation d'obfuscation *si* P *et* P' *possèdent le même comportement observable. Plus précisément, dans le but que* $P \xrightarrow{T} P'$ *soit une transformation d'obfuscation légale, les conditions suivantes doivent être respectées :*

- *si P échoue à terminer ou termine avec une condition d'erreur, alors P' peut ou ne peut pas terminer,*
- *sinon, P' doit terminer et produire la même sortie que P.*

Afin de caractériser la « force » de cette transformation, Collberg, Thomborson et Low définissent une métrique et plusieurs critères. Cette métrique, notée m, est définie sur l'intervalle des réels $[\,0\,;1\,]$, de telle sorte que plus la valeur de m tend vers 1, plus le résultat de la transformation est « fort ». Le calcul de m n'est pas clairement défini. Les critères suivants permettent d'évaluer la qualité de l'*obfuscation*, mais ne permettent pas de créer de nouvelles méthodes d'obfuscation. Nous les rappelons brièvement ici :

- ***efficacité*** (*effectiveness*) : annihiler l'obfuscation nécessite plus de ressources que sa mise en place ;
- ***puissance*** (*potency*) : n'affaiblit pas les autres méthodes d'obfuscation ;
- ***discrétion*** (*stealth*) : se compose de deux parties :
  - *discrétion locale* (*local stealth*) : un adversaire ne peut pas trouver où la transformation est appliquée,
  - *discrétion stéganographique* (*steganographic stealth*) : un adversaire ne peut pas savoir si la transformation a eu lieu ou non ;
- ***coût*** (*cost*) : surcoût pour l'exécution du programme en temps et en mémoire, après transformation.

In fine, pour Collberg, Thomborson et Low, le résultat de la transformation doit satisfaire les trois propriétés suivantes :

- comportement identique entre le programme source et le programme obfusqué,
- un programme obfusqué doit être utilisable et cacher quelque chose de calculable,
- un adversaire doit rencontrer des difficultés pour comprendre le programme obfusqué.

Nous terminons avec les travaux de Barak *et al.* [Bar+01], qui ont permis une avancée majeure sur les liens existant entre l'*obfuscation* et la cryptographie [2] :

Définition 2.3 [Bar+01] *Soit un programme P. Un obfuscateur $\mathcal{O}$ est une transformation de compilation de P notée $\mathcal{O}(P)$ qui satisfait les trois propriétés suivantes :*

2. Notons que la définition originale est définie pour un circuit et pour des relations probabilistes, mais qu'elle reste équivalente pour un programme P quelconque.

- ***fonctionnalité*** *(functionality) : $\mathcal{O}(P)$ calcule la même fonction que P (fournis le même comportement observable) ;*
- ***ralentissement polynomial*** *(polynomial slowdown) : pour tout programme P, le temps d'exécution de $\mathcal{O}(P)$ est, au mieux, plus lent d'un facteur polynomial que le temps d'exécution de P, ou d'un facteur polynomial plus grand que la taille de P ;*
- ***boîte noire virtuelle*** *(virtual black box) : tout ce qui peut être efficacement calculé avec $\mathcal{O}(P)$ peut être aussi calculé avec seulement un jeu d'entrées/sorties (oracle access) du programme P.*

Avec cette définition, Barak *et al.* montrent qu'il est toujours possible d'extraire au moins un bit d'information. Contrairement à la définition précédente, ici l'adversaire est clairement défini à l'aide du *paradigme de simulation* (*simulation paradigm*). Une *simulation* est un algorithme qui essaie de simuler l'interaction de l'adversaire avec une partie honnête, sans connaître les entrées privées de cette partie honnête [Bar01 ; Lin17].

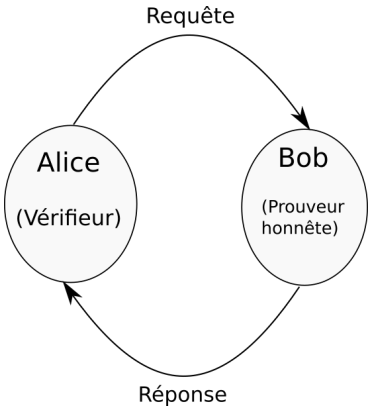


Figure 2.1 – Paradigme de simulation

Ce paradigme apparaît par exemple, lorsque deux parties Alice (le vérifieur) et Bob (le prouveur honnête) souhaitent interagir (par un jeu de requêtes et de réponses), et que Bob possède un secret. Ici, pour simplifier, nous dirons que le secret est un programme. Idéalement, toutes les interactions réalisées durant cette simulation ne devraient pas donner un avantage à Alice pour connaître le secret de Bob (figure 2.1). Or, le vérifieur peut toujours réussir à extraire un bit d'information d'un programme obfusqué. Alors, le temps nécessaire pour extraire des informations d'un programme obfusqué est corrélé à la capacité du vérifieur à réaliser une simulation capturant le comportement du programme. Nous venons de voir que l'utilisation d'une méthode d'obfuscation ajoute du bruit pour rendre l'interprétation d'un programme plus

longue. La capacité de l'adversaire à enlever ce bruit va dépendre de sa capacité à interpréter ce programme, autrement dit de sa capacité à se représenter ce programme. Pour un même programme, différentes représentations existent, et c'est ce que nous allons voir maintenant.

2.1.1.2 Représentations d'un programme

Un programme obfusqué est interprétable au moins par une machine cible et par une simulation réalisée par un adversaire. Plus généralement, un programme contient un ensemble d'informations calculables par une machine et interprétables dans différents systèmes de preuves, au moyen de différentes logiques. Chaque système de preuve peut être instrumentalisé par un adversaire pour réaliser un simulateur. Chaque simulateur peut être calculable. Nous souhaiterions pouvoir caractériser le choix du *modèle de calculabilité* (*computational model*) dans lequel les programmes obfusqués peuvent être exécutés ou simulés par l'adversaire. Ces informations nous permettent de connaître le niveau de *complexité calculatoire* (*computational complexity*) de chacun de ces programmes (avant et après transformation). Autrement dit, caractériser chaque transformation en fonction de la classe de programmes visée devient possible. En pratique, les travaux de Horváth et Buttyán [HB15] proposent de limiter cette caractérisation aux programmes représentés par une structure uniforme (par exemple reconnaissables par une *machine de Turing*) ou par une structure non-uniforme (par exemple reconnaissables par un circuit booléen, mais pas par une *machine de Turing*). En effet, tous les langages pouvant être décidés pour des circuits, les programmes représentables par une structure non-uniforme ne contiennent pas forcément d'algorithme permettant de résoudre un problème. Par exemple, un circuit contenant un capteur ne peut pas être représentable par une *machine de Turing*. Dans le cas des programmes reconnaissables en temps polynomial (c'est-à-dire définis dans la classe de complexité descriptive *P/poly*, il existe une équivalence polynomiale entre ces deux représentations (uniforme et non-uniforme), sous conditions [KL80]. Autrement dit, si un programme est reconnaissable en temps polynomial, la représentation des algorithmes contenus dans le programme n'a pas d'importance vis-à-vis du calcul. Lorsque cette reconnaissance n'est pas possible en temps polynomial, la représentation des algorithmes contenus dans le programme peut influencer les actions de transformation des programmes et le résultat obtenu.

2.1.1.3 Limites des transformations

Un programme obfusqué doit répondre à deux limites techniques. La première concerne la limite de taille et de temps d'exécution du programme. La seconde concerne le problème de

l'extraction de l'information contenue dans le programme. Nous développons ces limites dans les paragraphes ci-après.

2.1.1.3.1 Taille d'un programme et Temps d'exécution. Le temps d'exécution d'un programme dépend de sa représentation et du temps d'exécution de cette représentation. Plus précisément, le temps d'exécution dépend d'un équilibre entre la description choisie et le temps de calcul de chaque transition. Par exemple, si un programme est considéré comme une chaîne de caractères reconnaissable par une *machine de Turing*, une manière d'améliorer sa description est d'augmenter la table de transition de la *machine de Turing*. Dans ce cas, un nouveau symbole peut permettre de remplacer un ensemble de caractères par ce nouveau symbole, ce qui accélère l'exécution du programme. Nous rappelons qu'un programme obfusqué est un programme original avec des informations supplémentaires. Pour qu'un programme obfusqué conserve un temps d'exécution très proche du programme orignal, deux solutions sont disponibles. La première solution consiste à compresser le programme, de manière à ce qu'il soit toujours exécutable. Par exemple, la technique de *virtualisation* (*virtualization obfuscation*) [Col+12] réussit cet exploit, en créant un programme fonctionnant avec des super-opérateurs. La deuxième solution est de remplacer une partie des transitions du programme par des extraits de code spécialement conçus pour que l'extraction d'information de ce code allonge le temps d'interprétation du programme. Ces extraits de code doivent être suffisamment compressés pour que l'interprétation soit obligée de réaliser un nombre important d'opérations intermédiaires.

2.1.1.3.2 Le problème de l'extraction d'information. Les travaux de Ceccato *et al.* [Cec+17] montrent qu'en pratique, le concept peu contraignant de l'*obfuscation* dépend fortement de la *puissance* de l'attaquant, c'est-à-dire sa capacité à extraire de l'information dans un programme. Cette capacité est à mettre en parallèle avec la technique de rétro-ingénierie choisie. En effet, l'attaquant doit savoir distinguer clairement si ses outils et si la technique choisie pour effectuer son interprétation sont adaptés pour le programme observé. En d'autres termes, l'attaquant doit pouvoir répondre clairement à ces deux questions (décrites plus en détail ci-dessous) : « *Est-ce que ce que je sais, je le sais ?* » et « *Quelle méthode de preuve automatique utiliser ?* ». La première question est relative à l'état de connaissance de l'attaquant du programme et de l'environnement d'exécution du programme. La seconde question est relative au simulateur utilisé par l'attaquant. Ces deux questions peuvent être reliées à un problème plus général défini par Tarski [Tar72] : le problème de l'existence d'une différence entre une

inférence en lien avec un objet et l'interprétation permettant de juger de la vérité [3] de cette inférence. Nous développons maintenant les deux questions :

1. *« Est-ce que ce que je sais, je le sais ? »* [BN15] : l'attaquant va essayer de capturer des connaissances au moyen de diverses techniques avancées, de manière plus ou moins parfaite, pour être le plus efficace, c'est-à-dire avec un certain niveau d'approximation. Ces approximations peuvent amener le raisonnement effectué par l'adversaire à croire des choses qui n'existent pas, sans que ce dernier ne s'en rende compte.
2. *« Quelle méthode de preuve utiliser ? »* [BP01] : la puissance de l'attaquant est équivalente à sa capacité à pouvoir effectuer correctement une simulation du programme, c'est-à-dire à la quantité de connaissances en sa possession. Les informations manquantes peuvent être retrouvées à l'aide d'un système de raisonnement automatique, si ce dernier est assez puissant pour les capturer.

Nous venons d'introduire quelques concepts et limites importants de l'*obfuscation*. Pour nous aider à mieux comprendre comment ces idées se sont mises en place, nous proposons un petit historique sur l'histoire contemporaine de l'*obfuscation*.

2.1.2 Un peu d'histoire

L'histoire de l'*obfuscation* est à mettre en parallèle avec celle de la cryptographie. L'objectif de ces deux domaines est d'assurer une forme de confidentialité de l'information en transformant des informations sensibles en un format non compréhensible pour un adversaire pendant un temps limité. Nous proposons dans la figure 2.2 une chronologie simplifiée de l'évolution technique de l'*obfuscation*, mis en parallèle de quelques événements importants. Les *principes de Kerckhoffs* permettent d'énoncer des conditions simples pour l'utilisation d'un algorithme permettant de protéger de l'information. L'analyse automatique d'un programme, à l'aide d'une théorie du premier ordre, est un tour de force contemporain de plusieurs domaines de recherche, faisant suite à de nombreuses publications (Logique des prédicats, Théorie des modèles, Problème de l'arrêt, Théorie de l'Information, Calculabilité, . . .). Alors que la cryptographie se développa principalement à l'aide d'outils algébriques, l'utilisation de l'*obfuscation* se développa plus tardivement avec d'autres outils abstraits. Nous retrouvons par exemple les premières techniques d'obfuscation logicielle seulement à partir des années 1980. Ces méthodes d'obfuscation assez simples sont proposées par des développeurs de logiciels pour ralentir les copies de programmes

3. Le critère de vérité correspond à la non-contradiction d'une inférence.

illégales. Les premiers travaux académiques sur ce sujet démarrèrent dans les années 1990, quelques résultats importants furent démontrés dans les années 2000, et ce n'est qu'à partir des années 2010 qu'une explosion de publications apparut sur ce sujet [Gar+16 ; Xu+17]. Deux événements majeurs sont à l'origine de ce coup de projecteur : le premier est la mise en place du *Cyber Grand Challenges* (*CGC*) [SA15] organisés par la *DARPA* [4], et le second grâce aux premières publications de candidats respectant l'*obfuscation indistinguable* (*iO*) compatible avec différents schémas de chiffrement à clé publique [SW14]. Le premier événement permit de mettre un coup d'accélérateur sur les méthodes ou outils d'analyse de programmes. Le second permit un florilège de propositions de pistes pour créer une nouvelle génération de système de chiffrement. Au moment de la rédaction de ce manuscrit, le rythme de nouvelles publications a diminué, car les efforts pour avancer dans ce domaine deviennent de plus en plus importants.

2.1.3 L'obfuscation du graphe de flot de contrôle

Le flot de contrôle d'un programme présente les *dépendances de contrôle* (*control dependencies*) des différentes instructions d'un programme exécuté. Elles sont parfois représentées sous la forme d'un graphe dirigé appelé *graphe de flot de contrôle* (*control-flow graph* (*CFG*)). Chaque nœud d'un *CFG* représente une portion de code sans saut (appelé bloc de base (*basic block*)) et chaque arête représente un saut depuis un bloc de base vers un autre bloc de base. Un *CFG* permet de représenter tous les chemins d'exécutions qu'un programme peut emprunter. La transformation d'*étalement du flot de contrôle* (*control flow flattening*) défini par Wang *et al.* [Wan+00], a pour but de redéfinir la structure du *graphe de flot de contrôle* d'un programme, pour rendre obscurs les liens entre les différents *blocs de bases* (*basic block*). L'astuce utilisée est la suivante : réussir à transformer les informations du *flot de contrôle* (*control flow*) dans le *flot de données* (*data flow*). Nous présentons ici les deux applications actuelles de cette transformation. La première utilise un « dispatcheur » central [Cho+01]. La seconde permet de remplacer les sauts et appels directs [LD03]. Cette transformation peut être renforcée à l'aide de *prédicats opaques* [CP10] et de fonctions de hachage cryptographique dont la *graine* (*seed*) est un vecteur de *prédicats opaques*.

4. DARPA : *Defense Advanced Research Projects Agency* (*Agence (américaine) pour les projets de recherche avancée de défense*)

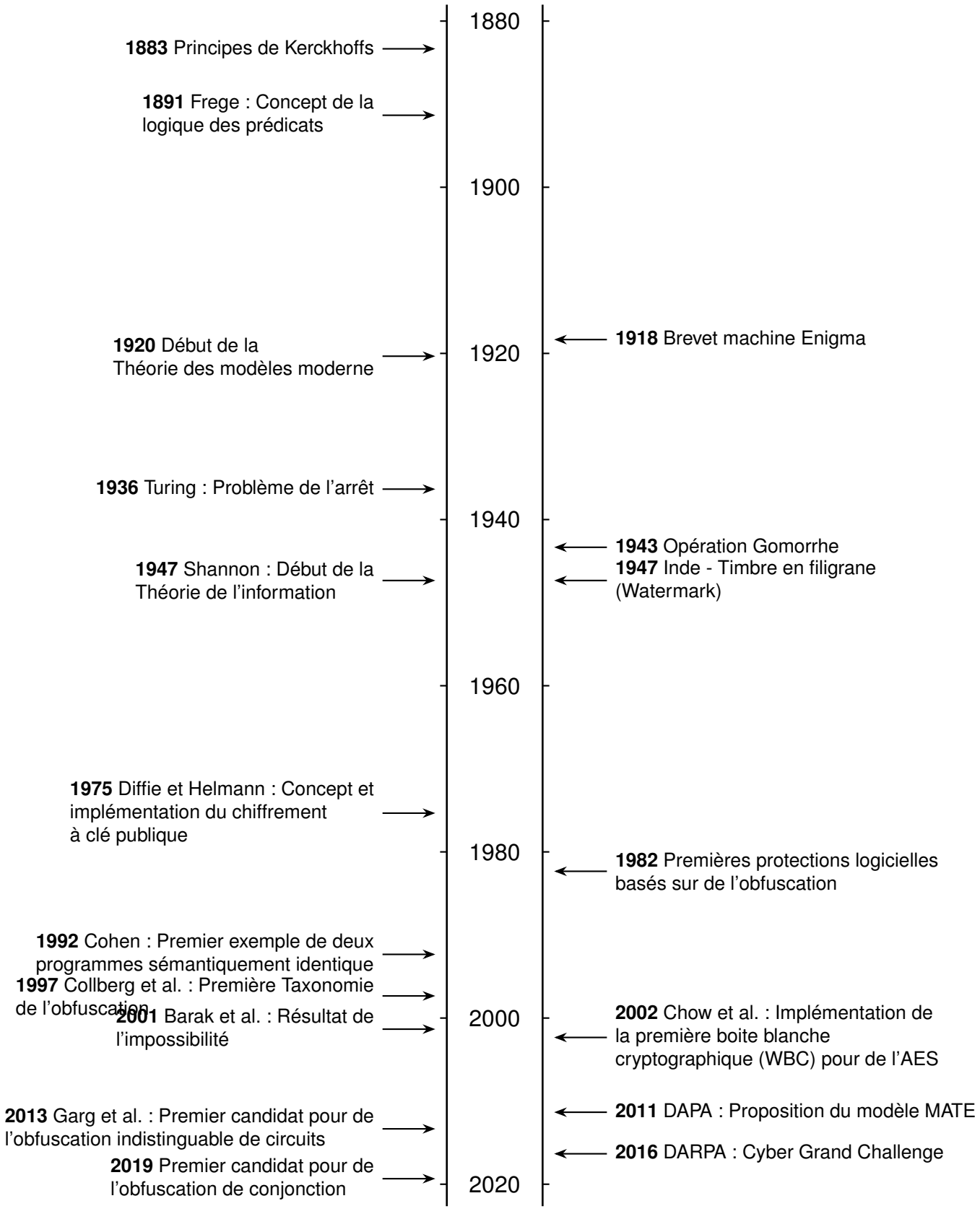
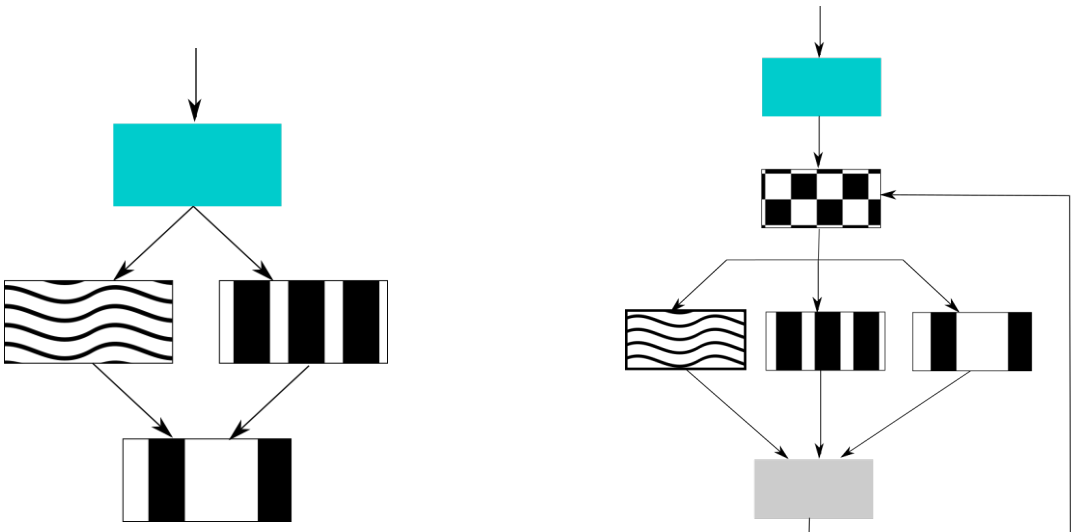


FIGURE 2.2 – Chronologie simplifiée sur les évolutions techniques de l'obfuscation

2.1.3.1 Utilisation d'un « dispatcheur » central

Pour un programme simple, une transformation naïve de l'étalement du flot de contrôle pourrait être la suivante. Le *flot d'exécution* (*execution flow*) sera dirigé à l'aide du « dispatcheur », de manière à ce qu'il indique quel est le prochain bloc de codes à exécuter. Dans ce but, la nouvelle variable est mise à jour à la fin de chaque bloc, de façon à indiquer le prochain bloc. Le flot d'exécution revient vers le dispatcheur qui va être dirigé vers un nouveau bloc. La figure 2.3 présente un exemple d'utilisation de cette méthode.



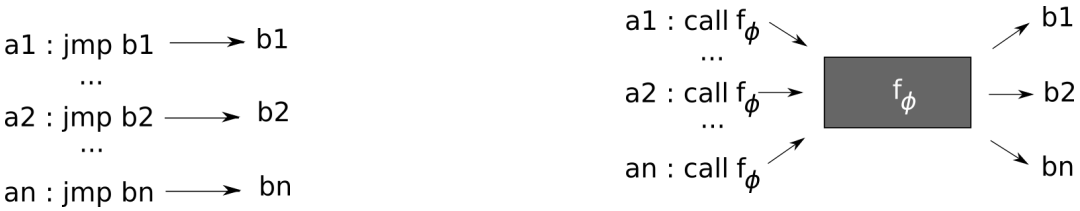
(a) *CFG* d'un programme simple

(b) *CFG* du même programme après étalement

FIGURE 2.3 – Obfuscation du graphe de flot de contrôle avec un « dispatcheur » central

2.1.3.2 Remplacement des sauts directs

Les *fonctions branches* (*branch function*) sont des fonctions qui ne reviennent pas à leur appelant. Elles transfèrent le *flot de contrôle* à une adresse différente calculée depuis l'adresse de retour et d'un *décalage* (*offset*) donné comme paramètre à la fonction branche. Ainsi un saut direct peut être remplacé par un *appel* (*call*) et un *saut indirect* (*indirect jump*) basé sur les arguments de l'appel. La figure 2.4 illustre une manière d'appliquer cette méthode d'obfuscation.



(a) Programme original

(b) Une fonction branche f de paramètre ϕ

FIGURE 2.4 – Obfuscation du graphe de flot de contrôle avec remplacement des sauts directs

Bilan

L'*obfuscation* d'un programme est une tâche difficile à mettre en œuvre, car cela suppose de mettre en place une méthode de transformation d'un programme qui ajoute du bruit pour une ou plusieurs interprétations d'un programme, sans changer son temps d'exécution. Nous avons vu qu'il existe différentes pistes pour comprendre ce qu'est l'*obfuscation*. Nous nous concentrons dans ce manuscrit sur les méthodes permettant de rendre plus difficile l'interprétation du *graphe de flot de contrôle* d'un programme. Nous définissons dans la prochaine section les différentes propriétés des modèles de sécurité et d'attaquants pour l'*obfuscation*.

2.2 Modèles de sécurité et d'attaquant

Nous regardons ici quelles sont les propriétés permettant de garantir la sécurité attendue d'une *obfuscation* au travers de deux modèles de sécurité. Dans ces modèles, l'attaquant est limité par la puissance de calcul et la quantité de temps disponible. Pour une vision plus réaliste de l'attaquant, nous regardons ensuite deux modèles d'attaquant.

2.2.1 Modèles de sécurité

2.2.1.1 Sécurité par simulation (*VBB*)

Une *boîte noire* (*black box*) est un concept permettant de représenter un système pour ses couples d'entrées et de sorties, indépendamment de ses mécanismes internes.

La propriété de *boîte noire virtuelle* (*virtual black box (VBB)*), définie par BARAK *et al.* [Bar+01], suppose qu'un programme se comportant comme une *boîte noire* respecte le modèle de sécurité par simulation. La notion de *virtuel* (*virtual*) (definition 2.4) est le terme mathématique pour dire qu'un objet se comporte de la même manière qu'un autre objet. Dans ce modèle,

un attaquant qui observe le comportement d'un programme obfusqué $\mathcal{O}(P)$ ne peut distinguer clairement la simulation de ce programme du véritable programme. Autrement dit, l'attaquant ne peut faire mieux que de reproduire les couples d'entrées et de sorties du programme obfusqué. La figure 2.5 représente ce modèle de sécurité : où le programme P est un programme reconnaissable par une *machine de Turing* ou un circuit booléen. Par exemple, un adversaire observant un système de mot de passe obfusqué respectant ce modèle de sécurité ne peut pas faire mieux que d'exécuter une attaque par dictionnaire, selon Wagner et Goldberg [WG00].

Définition 2.4 *Un objet mathématique a virtuellement la propriété X s'il possède un sous-objet fini (sous-groupe d'index fini) possédant la propriété X.*

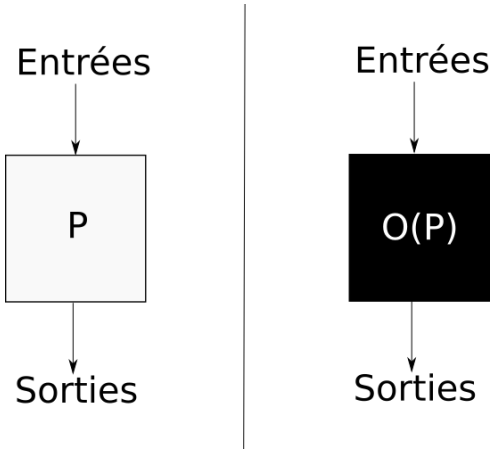


Figure 2.5 – Modèle de sécurité par simulation (*VBB*)

2.2.1.2 Sécurité basée sur l'indistinguabilité (*iO*)

Le modèle de sécurité basé sur l'indistinguabilité suppose qu'un adversaire ne peut pas distinguer, d'un point de vue du calcul (*computationally indistinguishable*), deux versions équivalentes (en temps et en mémoire) d'un même programme et réalisant les mêmes fonctions. Dans ce modèle, l'adversaire est autorisé à simuler les opérations internes à l'aide d'un oracle, sans restriction sur le nombre de requêtes échangées avec cet oracle, selon Brakerski et Rothblum [BR14b]. Pour que ce modèle puisse être réalisable sur une machine, il existe au moins deux conditions à respecter. La première, proposée par Garg *et al.* [Gar+13], est que le programme soit défini au plus dans la classe de complexité NC_1 (la classe des circuits de taille polynomiale P et calculable en parallèle [5]). La seconde, proposée par Goldwasser et Rothblum

5. `https://complexityzoo.uwaterloo.ca/Complexity_Zoo`

[GR14] montrent que les transformations appliquées à un programme ne sont pas définies pour les circuits définis dans la classe de complexité polynomiale P. Ces circuits doivent posséder au moins une affirmation supplémentaire (par exemple : la propriété de *sens unique* (*one-way property*) indique qu'un circuit demande beaucoup de ressources en temps et en calculs pour être inversé). La figure 2.6 présente ce modèle de sécurité, pour deux programmes P_1 et P_2, de mêmes tailles, et calculant la même fonctionnalité. L'*obfuscation* de chacun de ces programmes permet d'obtenir deux nouveaux programmes $\mathsf{O}(\mathsf{P}_1)$ et $\mathsf{O}(\mathsf{P}_2)$, qui sont respectivement les versions obfusqués de P_1 et P_2. Les deux programmes obfusqués sont indistinguables, c'est-à-dire qu'ils sont équivalents en taille et en temps d'exécutions.

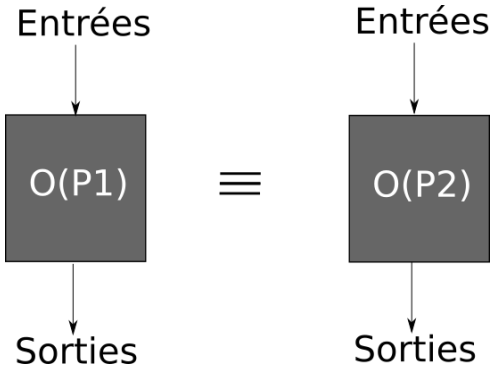


FIGURE 2.6 – Modèle de sécurité basé sur l'indistinguabilité (*iO*)

2.2.2 Modèles d'attaquant

2.2.2.1 Attaquant en boîte blanche cryptographique (*White-Box Cryptography*)

Dans le modèle d'attaquant en *boîte blanche cryptographique* (*White-Box Cryptography* (WBC)) [Cho+02], l'attaquant possède l'accès à l'ensemble des processus d'exécution et des variables d'environnement de programmes binaires. Son but est d'analyser l'implémentation d'un crypto système contenant des primitives cryptographiques et de retrouver les clés de ce système. Les valeurs de ces clés sont masquées dans l'implémentation. Le problème de la construction des primitives en boîte blanche sécurisées peut être vu comme un cas spécial de l'*obfuscation*.

2.2.2.2 Attaquant de l'homme à la fin (*MATE*)

Le modèle de l'attaquant de *l'homme à la fin* (*Man At The End* (MATE)) [Akh+15] considère que l'assaillant possède un accès direct à l'hôte où le programme doit être exécuté, c'est-

à-dire qu'il a tout pouvoir sur le programme final exécuté (accès physique et modifications du programme sans limites). Son but est d'extraire toutes les informations du programme, et de casser toutes les protections présentes [Cec+17]. L'utilisation de ce modèle permet de graduer la quantité d'effort qu'un humain doit effectuer, dans un temps imparti, pour extraire les informations contenues dans un programme, autrement dit de nommer les différentes étapes d'une attaque en fonction des ressources disponibles. En pratique, cela permet de comparer différentes méthodes d'*obfuscation* [Man+16]. Des exemples d'attaques correspondant à ce modèle peuvent être le vol de données propriétaires, ou les méthodes mises en place pour casser les processus de vérification de licences.

Bilan

Différents modèles de sécurité et d'attaquant existent pour l'*obfuscation*. L'utilisation de chaque modèle ne peut se faire que dans un contexte bien précis. Nous nous limitons maintenant à une méthode d'obfuscation : les prédicats opaques.

2.3 Les prédicats opaques

En tant que technique d'obfuscation, les *prédicats opaques* (*opaque predicates*) augmentent la complexité d'analyse du *CFG*. Ils ajoutent des chemins non suivis (*infeasible paths*), du code mort (*dummy code, dead code*) ou du tatouage numérique (*watermark*), afin d'augmenter les efforts nécessaires à leur identification et à la déobfuscation. De nombreux outils d'obfuscation intègrent cette technique dans leur chaîne de transformations comme `Epona` [Bru+19], `Obfuscator-LLVM` [Jun+15], `Tigress` [Col+19], `VMProtect` [20c], `PELock` [20b], `ASProtect` [20a]. Dans ce manuscrit, nous abordons la description et l'analyse de *prédicats opaques*. Nous n'aborderons pas les méthodes de fabrication des prédicats opaques pour de l'obfuscation de code, qui à ce jour restent un sujet ouvert. Nous proposerons seulement une piste de conception de prédicats opaques, dans le chapitre 7. Comme nous allons le voir dans les paragraphes suivants, le terme *opaque* pour un prédicat est une définition ouverte. Ce concept indique qu'un attaquant devra faire plus d'effort pour interpréter l'expression dans le but de connaître son résultat. En pratique, les prédicats opaques sont difficiles à détecter. Pour augmenter cette difficulté, les prédicats sont combinés avec d'autres méthodes, comme les fonctions de hachages, la virtualisation, ou l'encodage des variables.

2.3.1 Définitions et limites

Nous nous intéressons à la classe de programmes pouvant être traduite en une formule définie en logique équationnelle du premier ordre. Dans ce cas, les formules obtenues sont des ensembles de clauses qui se décomposent en deux parties. La première est un ensemble de clauses définissant les propriétés des programmes (par exemple les propriétés de structures de données : liste, tableaux). La seconde est un ensemble de clauses qui traduisent le comportement du programme. Nous présentons maintenant les fonctions mathématiques qui utilisent des prédicats, et qui sont naturellement adaptées pour utiliser des *prédicats opaques*. Ces fonctions mathématiques permettent une première modélisation simple des conditions de branchement d'un programme. Puis nous présentons les *prédicats opaques* et leurs limites actuelles.

2.3.1.1 Les fonctions point

Les *fonctions point* (*point functions*) correspondent à une représentation formelle d'une condition de branchement dans un *CFG*. Nous définissons deux familles de fonctions point : les fonctions point à un bit de sortie et les fonctions point à plusieurs bits de sortie.

Les fonctions point à un bit de sortie (PF). Une *fonction point à un bit de sortie* (*point function with a bit output*), ou plus simplement, s'il n'existe pas d'ambiguïté, une *fonction point* (*point function (PF)*) [6], est une fonction booléenne $F_{\mathtt{x}} \colon \{\,0\,,1\,\}^* \to \{\,0\,,1\,\}$, tel qu'il existe une entrée unique $\mathtt{k}$ de taille n, pour laquelle la fonction retourne la valeur 1, lorsque $\mathtt{k}$ est égal à une valeur $\mathtt{x}$, sinon la fonction retourne la valeur 0 (*c.f.* équation 2.1).

$$\forall k \in \{\,0\,,1\,\}^*, \exists ! \mathtt{x} \in \{\,0\,,1\,\}^n, F_{\mathtt{x}}(k) = \begin{cases} 1 & \text{si } k = \mathtt{x} \\ 0 & \text{sinon} \end{cases} \tag{2.1}$$

Nous regardons maintenant l'utilisation des *fonctions point*, parfois utilisées pour cacher les *mots de passe* (*password-hiding algorithms*) [Can97 ; CMR98]. Pour une chaîne de caractère $\mathtt{x}$ de taille n, une *fonction point* $F_{\mathtt{x}} \colon \{\,0\,,1\,\}^* \to \{\,0\,,1\,\}$ contenue dans un programme P peut être représentée par un circuit $C_{\mathtt{x}'} \colon \{\,0\,,1\,\}^m \to \{\,0\,,1\,\}$, avec $\mathtt{x}'$ une chaîne de caractère de taille n' telle que $\mathtt{x}$ et $\mathtt{x}'$ sont équivalentes, et m la taille maximum de la représentation des éléments dans ce circuit. L'ensemble $\{\,0\,,1\,\}^m$, c'est-à-dire l'ensemble des chaînes de caractères de taille au plus m, doit être équivalent à un sous-ensemble de $\{\,0\,,1\,\}^*$. Autrement dit, chaque élément défini dans l'ensemble $\{\,0\,,1\,\}^m$ possède un élément équivalent dans l'ensemble $\{\,0\,,1\,\}^*$. Par

6. En mathématique, la fonction point est équivalente dans le domaine discret au delta de Kronecker.

définition, ces ensembles sont non uniformes, c'est-à-dire que la taille des éléments définis dans ces ensembles n'est pas constante. La fonction $F_{\mathtt{x}}$ (resp. le circuit $C_{\mathtt{x}}$) retourne la valeur 1 pour la valeur d'entrée $\mathtt{k}$ égale à $\mathtt{x}$ (resp. $\mathtt{k}'$ équivalente à $\mathtt{x}'$), la valeur 0 pour les autres entrées (*c.f* équation 2.2).

$$\forall k' \in \{\,0\,,1\,\}^m, \exists!\mathtt{x}' \in \{\,0\,,1\,\}^{n'}, C_{\mathtt{x}'}(k') \leftarrow \begin{cases} 1 & \text{si } k' \equiv \mathtt{x}' \\ 0 & \text{sinon} \end{cases} \tag{2.2}$$

Lorsqu'un attaquant utilise un simulateur pour comprendre le comportement d'un programme calculant une *fonction point*, l'attaquant vérifie en fait si le domaine de définition de la *fonction point* simulée est bien l'ensemble $\{\,0\,,1\,\}^*$ [Can97]. Alors, dans ce cas, l'attaquant peut trouver la chaîne de caractère $\mathtt{x}$. Sinon le simulateur retourne une chaîne de caractères aléatoire, définie dans l'ensemble $\{\,0\,,1\,\}^m$. Autrement dit, le simulateur parcours de manière aléatoire l'ensemble des solutions, de la même manière qu'une *marche aléatoire* (*random walk*) sur un *graphe expanseur* (*expander graph*) défini par Goldreich [Gol11].

Plusieurs solutions génériques existent pour l'obfuscation d'une *fonction point* dans nos deux modèles de sécurité. Nous retenons ici la solution en lien avec le concept de prédicat pour la *fonction point*, car plus adaptées à notre cas d'étude. Pour le modèle *VBB* (section 2.2.1), la *fonction point* doit posséder la propriété de sens unique (définition 2.5) [Wee05], et pour le modèle d'obfuscation indistinguable (iO) [BS16], la *fonction point* doit de plus être *injective*.

Définition 2.5 [Wee05] *Une fonction point est dite à* sens unique *(one-way) si elle est représentable par des* permutations [7]*, telle que pour inverser une permutation, un attaquant ne peut pas faire mieux que d'analyser cette permutation en boîte noire.*

Les fonctions point à plusieurs bits de sortie (MBPF). Une extension naturelle des fonctions point concerne les *fonctions point avec plusieurs bits de sorties* (*point function with multibits output* (MBPF)). Une fonction *MBPF* $F_{\mathtt{x},\mathtt{y}} \colon \{\,0\,,1\,\}^* \rightarrow \{\,0\,,1\,\}^m$ permet de généraliser les fonctions point, vues précédemment, en permettant le retour d'une chaîne de caractères de taille $\mathtt{m}$, au lieu d'un booléen (*c.f.* équation 2.3).

$$\forall k \in \{\,0\,,1\,\}^*, \exists \mathtt{y} \in \{\,0\,,1\,\}^m, \exists!\mathtt{x} \in \{\,0\,,1\,\}^n, F_{\mathtt{x},\mathtt{y}}(k) = \begin{cases} \mathtt{y} & \text{si } k = \mathtt{x} \\ 0 & \text{sinon} \end{cases} \tag{2.3}$$

7. Les permutations peuvent être représentées par des fonctions bijectives.

$$\forall k' \in \{0,1\}^a, \exists \mathtt{y}' \in \{0,1\}^b, \exists! \mathtt{x}' \in \{0,1\}^c, F_{\mathtt{x}',\mathtt{y}'}(k') \leftarrow \begin{cases} \mathtt{y}' & \text{si } k' \equiv \mathtt{x}' \\ 0 & \text{sinon} \end{cases} \tag{2.4}$$

L'existence des fonctions *MBPF* permet de réaliser l'obfuscation de la composition de *fonctions point* [CD08]. Malheureusement, les transformations appliquées pour obtenir une fonction point obfusquée ne peuvent pas être appliquées de la même manière que sur une fonction *MBPF* [Can+10]. Ainsi, pour qu'une fonction *MBPF* puisse être obfusquée, la distribution des chaînes de caractères des *fonctions point* (*PF*) utilisées doit être la plus uniforme possible [Can+10]. Autrement dit, une fonction MBPF est définissable dans un modèle de sécurité si la probabilité suivante « une chaîne de caractères d'une *fonction point* (*PF*) existe dans l'ensemble des chaînes de caractères » tend vers la même valeur pour chaque chaîne de caractères. Dans ce cas, un attaquant cherchant à simuler la fonction MBPF obfusquée pour déterminer une chaîne `x` observera globalement un nombre de requêtes envoyées à l'oracle uniforme. Autrement dit, pour déterminer chaque chaîne de caractères, l'attaquant devra échanger quasiment le même nombre de requêtes avec l'oracle [Can+10].

2.3.1.2 Les prédicats opaques

En informatique, un prédicat `P` est une expression booléenne représentée par la fonction : $P\colon X \to \{0,1\}$, où X est un ensemble fini, et $\{0,1\}$ est un ensemble à deux éléments. L'élément 1 correspond à l'évaluation du booléen VRAI ($\top$), et l'élément 0 à l'évaluation du booléen FAUX ($\bot$). Un prédicat possède la propriété *constante* si sa fonction P retourne toujours la même image, c'est-à-dire que pour tous les éléments appartenant à l'ensemble X, l'élément retourné par la fonction est toujours le même. Autrement dit, un prédicat constant est une expression propositionnelle retournant toujours la même valeur de vérité (VRAI ou FAUX). Ce comportement peut être représenté par un autre prédicat qui est toujours VRAI, c'est-à-dire par une tautologie. Nous définissons maintenant les *prédicats opaques* et ses principales sous-familles.

Prédicat opaque Collberg, Thomborson et Low [CTL97] donnent la première définition d'un prédicat opaque :

Définition 2.6 [CTL97] Un prédicat `P` est opaque *si à un point* `p` *d'un programme la sortie de ce prédicat est connue au moment de l'obfuscation (obfuscation time).*

Autrement dit, un *prédicat opaque* est défini par une fonction booléenne constante dont le

comportement constant est facile à réaliser au moment de l'exécution, mais difficile à comprendre par un attaquant. Les *prédicats opaques* ont un comportement similaire à une *fonction point* constante au moment de l'exécution. À notre connaissance, peu de méthodes sont connues pour fabriquer des *prédicats opaques* au niveau du binaire [MT06 ; SI14 ; Yad16], ou au niveau du code (source ou intermédiaire) [CTL97 ; Sha+08 ; DP08 ; Wan+11 ; Oll+19b]. De plus, malgré l'existence d'un nombre important de familles de *prédicats opaques*, la littérature actuelle ne propose pas assez de cas d'étude de prédicats opaques utilisés en obfuscation. Le lecteur pourra trouver des exemples de prédicats arithmétiques dans le tableau 2.1.

$\forall x, y \in I$	$7y^2 - 1 \neq x^2$
$\forall x, y \in I$	$x * (x + 1) = 0 \bmod 2$
$\forall x \in I$	$x^2 \geq 0$

TABLE 2.1 – Exemples de prédicats opaques arithmétiques

Exemple 1:
Un prédicat opaque en assembleur *x86* peut être le suivant :

```
1 xor eax, eax
2 jz loc_to_jump
```

Pour toute les exécutions, l'instruction `jz` (*jump if zero*) sautera toujours au label `loc_to_jump` car dans notre cas, l'indicateur de zéro (*zero flag*) est toujours fixé par l'instruction `xor`. L'indicateur de zéro est ici toujours évalué à la valeur 1 (c'est-à-dire VRAI), car l'instruction `xor` est toujours égale à zéro.

L'utilisation classique des *prédicats opaques* en *obfuscation* repose sur leur intégration dans une condition de branchement. Il existe trois manières différentes de les utiliser : statique, dynamique, ou aléatoire . Nous présentons maintenant chacune de ces utilisations.

Prédicat opaque statique [CTL97] Un prédicat opaque est dit *statique* lorsque son comportement reste identique pour toutes les exécutions du programme, c'est-à-dire qu'une seule branche est parcourue pour toutes les exécutions du programme. Pour que l'illusion fonctionne, il faut aussi ajouter dans le *bloc de base* (*basic bloc*) suivant du *faux code* (*dummy code*), ou code factice, qui ressemble sémantiquement à d'autres parties du programme, sans aucune utilité (figure 2.7).

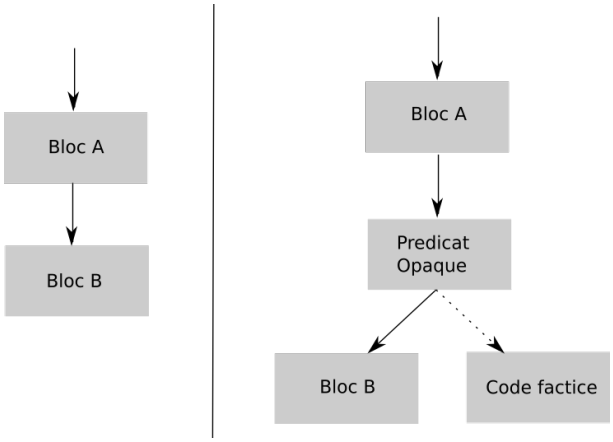


FIGURE 2.7 – Fonctionnement d'un prédicat opaque statique

Prédicat opaque dynamique [Pal+00] Un prédicat opaque est dit *dynamique* lorsque son comportement change pour différentes exécutions, mais qu'il oriente toujours vers le même code, bien que celui-ci change aussi dans sa forme pour chaque exécution (figure 2.8).

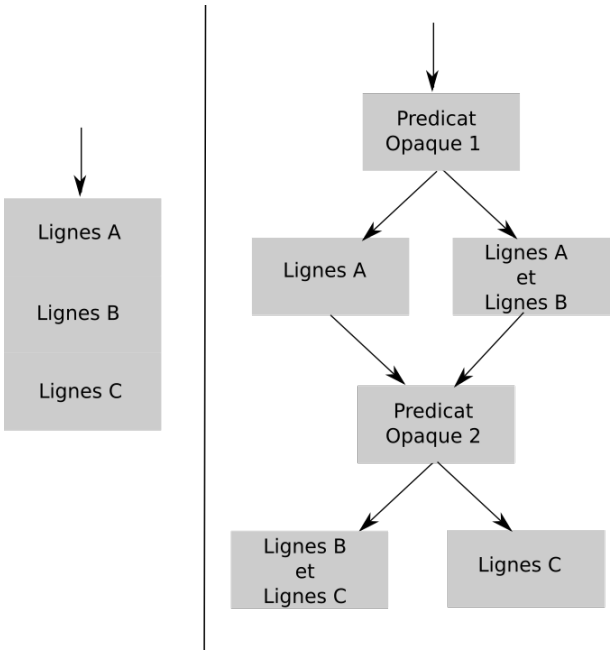


FIGURE 2.8 – Fonctionnement d'un prédicat opaque dynamique

Prédicat opaque aléatoire Les prédicats opaques *aléatoires*, aussi appelés *prédicats opaques à deux chemins* (*two-way opaque predicate*), sont une variante du prédicat opaque, où chaque

branche de la conditionne dirige vers du code syntaxiquement différent, mais équivalent fonctionnellement. Le but étant que l'attaquant perd du temps à comprendre que les deux blocs après le prédicat sont équivalents (figure 2.9).

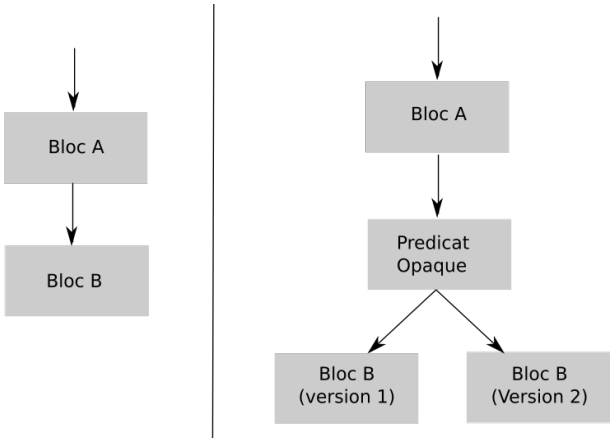


FIGURE 2.9 – Fonctionnement d'un prédicat opaque aléatoire

Prédicat opaque à trappe Les *prédicats opaques à trappe* (*trapdoor predicate*) sont des prédicats opaques avec un comportement légèrement différent. Ces prédicats sont constants pour toutes les valeurs d'entrées, sauf une. Ils sont alors équivalents à une condition de branchement classique, et sont difficilement distinguables d'un prédicat opaque statique.

2.3.1.3 Quelques limites

L'utilisation des *prédicats opaques* n'est envisageable dans un programme que sous certaines conditions :

- **un flot de contrôle riche** (*rich control flow*) [ZGR17] : Le programme original contient un ensemble de branchement conditionnel important ,
- **un générateur de faux code** (*dummy code generator*) [ZGR17] l'utilisation de ce générateur permet de créer du code qui soit sémantiquement indistinguable du reste du programme,
- **peu de perturbations** sur l'exécution et la quantité de mémoire utilisée par le programme original,
- **pas d'opérations dépendantes d'une bibliothèque** [CTL97], le programme ne fait pas appel à des fonctions définies dans des bibliothèques, le programme n'est constitué que de code natif.

Nous venons de voir les caractéristiques des *prédicats opaques*. Nous pouvons regarder maintenant les différentes méthodes de détection des *prédicats opaques*.

2.3.2 Méthodes de détections classiques

La méthode classique de résolution d'un *prédicat opaque* est proposée par Nagra et Collberg [NC09] :

1. localiser les instructions qui composent le prédicat opaque P,
2. déterminer les symboles correspondant aux variables d'entrées de P,
3. déterminer le domaine de ces entrées,
4. déterminer les sorties de P pour tous les arguments définis dans le domaine des entrées

Le point de départ dépend de la détection du *prédicat opaque*. Pour une attaque *MATE* (section 2.2.2), la détection de *prédicats opaques* s'appuie sur une détection humaine ou à l'aide d'outils d'analyse automatique [ZGR17]. Dans cette thèse, nous nous sommes restreints aux techniques automatiques. Les premiers travaux à ce sujet furent proposés par Madou, Van Put et De Bosschere [MVD06], en vérifiant par *fuzzing* (*fuzz testing*) avec un taux important d'erreur de potentiels candidats de *prédicats opaques statiques*.

Nous listons maintenant les différentes méthodes classiques automatisables, proposées par Zobernig, Galbraith et Russello [ZGR17], permettant la détection et l'analyse des *prédicats opaques*, nous les regroupons en deux classes : attaques statiques et attaques dynamiques. Nous y ajoutons aussi leurs avantages et leurs inconvénients d'un point de vue attaquant.

2.3.2.1 Attaques statiques

Les attaques statiques ont l'avantage de pouvoir donner des résultats précis et rapides, sans avoir besoin d'exécuter le code. Elles ont le désavantage de supposer que le *CFG* est facilement disponible. Elles ne permettent pas la prise en compte complet du *multithreading* ou de l'*automodification* du code.

- **recherche par force brute** (*brute force search*) : lorsque le domaine de définition X du prédicat est suffisamment petit, alors l'attaquant peut simuler le prédicat P en *boîte noire* pour obtenir tous les couples d'entrées et de sorties.

  Avantages : permets d'obtenir tous les couples d'entrées et de sorties.

Inconvénients : si le nombre d'éléments contenus dans le domaine de définition du prédicat est très important, il est probable qu'il ne soit pas possible en pratique de calculer tous les couples d'entrées et de sorties.

- **évaluation à zéro** (*evaluate at zero*) : en pratique, cette méthode semble suffisante pour dire si des prédicats constants sont possiblement des *prédicats opaques* toujours faux, en simulant la sortie du prédicat uniquement pour la valeur d'entrée 0. Si le prédicat retourne VRAI, alors le prédicat n'est pas un prédicat constant faux. La méthode ne suppose rien de plus. Si le prédicat retourne FAUX, la méthode indique avec un drapeau que le prédicat est possiblement opaque pour effectuer une analyse plus profonde éventuellement.

  Avantages : nécessite seulement la réalisation d'une simulation

  Inconvénients : l'hypothèse que le prédicat retourne toujours le booléen FAUX est faible, et le test ne permet que d'informer l'attaquant que le prédicat est peut-être opaque

- **test probabiliste** (*probabilistic check*) : pour un sous-ensemble uniforme du domaine de définition du prédicat, le simulateur regarde les sorties obtenues. Si ces sorties retournent toujours VRAI alors le prédicat est possiblement constant.

  Avantages : méthode peu contraignante dans sa mise en place

  Inconvénients : utilisation de plusieurs hypothèses faibles : le prédicat retourne toujours la valeur VRAI et l'ensemble du domaine de définition est uniforme. Si cette dernière hypothèse n'est pas vérifiée, plus le nombre d'éléments contenus dans cet ensemble est grand, plus le risque d'erreur est important.

- **détection de motif** (*pattern matching*) : l'attaquant compare l'expression du prédicat avec un ensemble de prédicats déjà connus.

  Avantages : méthode très efficace, car peu de risques d'erreurs.

  Inconvénients : l'obtention des motifs peut être difficile. Les motifs inconnus ne sont pas couverts.

- **preuves automatiques** (*automated proving*) : l'attaquant simule le prédicat à l'aide d'un *solveur SMT* pour montrer qu'il est constant. Si le solveur ne trouve pas de contre-exemple, alors la propriété recherchée est vérifiée.

  Avantages : le solveur permet d'avoir une « preuve » que le prédicat est constant.

  Inconvénients : la fabrication du modèle capturant le comportement du prédicat peut être difficile.

Nous définissons maintenant les méthodes de détection dynamiques des prédicats opaques.

2.3.2.2 Attaques dynamiques

Les attaques dynamiques peuvent traiter du code contenant du code *automodifiant*, du *multi-threading* et permettre l'identification des fonctions cryptographiques [CFM12]. Par contre, elles ne permettent pas d'obtenir la couverture maximale de code, c'est-à-dire la quantité maximale de code exécutable présent dans le programme.

- **analyse de teintes** (*taint analysis*) : l'attaquant regarde l'utilisation des blocs de code sélectionnés par l'expression conditionnelle. Si un bloc n'est jamais parcouru, alors l'expression conditionnelle est un *prédicat opaque*.

  Avantages : détection rapide du prédicat opaque

  Inconvénients : les prédicats opaques dynamiques ne sont pas détectés

- **traces d'exécution** (*execution traces*) : l'attaquant exécute dans un environnement contrôlé des parties du programme et enregistre les valeurs de tous les prédicats.

  Avantages : détection rapide du prédicat opaque

  Inconvénients : l'attaquant doit aussi comparer les instructions après le prédicat opaque

2.3.3 Autres travaux associés

Nous venons de voir différentes méthodes de détections des *prédicats opaques*. Ces derniers sont rarement utilisés seuls et sont utilisés dans des contextes très différents. Pour donner une idée de leur utilisation, nous présentons une technique de renforcement des prédicats et une liste de leurs applications.

2.3.3.1 Renforcement des prédicats à l'aide d'une fonction de hachage

Les *prédicats opaques* peuvent être composé avec des fonctions de hachages [Sch+16], cryptographiques ou non, pour renforcer leur opacité, face à de l'analyse symbolique. Par exemple, l'expression conditionnelle $p(x) = k$ devient avec l'utilisation d'une fonction de hachage h, $h(p(x)) = h(k)$. Nous remarquons que l'intérêt principal d'utiliser des fonctions de hachage n'est pas forcément sa propriété de *sens unique*, mais sa version plus faible, qui est la propriété d'*effet avalanche* (*avalanche effect*). Cette propriété permet d'augmenter le nombre d'étapes nécessaire pour inverser la fonction de hachage. Dans le cas d'une analyse symbolique, un prédicat ayant cette propriété demandera un effort d'interprétation plus important que le même prédicat sans cette propriété.

2.3.3.2 Des applications dans de nombreux domaines

Depuis les premières propositions de Collberg, Thomborson et Low [CTL97], les prédicats opaques sont utilisés dans de nombreux domaines pour leur faible coût et leur discrétion. Nous proposons ici au lecteur une liste non exhaustive de ces applications :

- mutation de logiciel malveillant métamorphique (*metamorphic malware mutation*) [BMM06 ; BMM07 ; BDM17]
- tatouage numérique (*software watermarking*) [HD11]
- techniques de désynchronisation de désassemblage de binaire [SH12 ; Gab14]
- détection de plagiat de logiciel [Luo+14]
- régulateur d'une valeur de cryptomonnaie [Che+17]

Bilan

Nous venons d'introduire les *fonctions point* et les *prédicats opaques*. Nous venons de voir qu'il existe différentes familles de prédicats opaques, et qu'il existe aussi différentes méthodes pour les détecter. Il existe très peu de contraintes pour fabriquer des *prédicats opaques*, mais il n'existe pas d'algorithme permettant de les fabriquer automatiquement. Les *prédicats opaques* peuvent être renforcés à l'aide d'autres fonctions pour ajouter des propriétés au prédicat. Les prédicats opaques sont utilisés dans de nombreux domaines.

2.4 Conclusion du chapitre

Dans ce chapitre nous avons eu l'occasion de voir que l'*obfuscation* est un concept décrit par ses effets sur l'interprétation d'un programme : allonger le temps de l'interprétation. Bien que le concept d'*obfuscation* ne vient pas de l'informatique, l'augmentation récente du nombre d'appareils électronique en fait une solution à bas coût pour protéger les logiciels le temps de leurs durées de vie. Différents modèles d'attaquants et de sécurité définissent un contexte d'utilisation de l'*obfuscation*. Malheureusement, il n'existe pas encore d'algorithme permettant de respecter strictement ces modèles. Nous proposons de regarder une des méthodes d'*obfuscation* réalisée par des *prédicats opaques*. Les *prédicats opaques* sont une solution facile à mettre en œuvre dans un programme pour ralentir l'interprétation du *graphe de flot de contrôle* d'un programme. Différentes familles de *prédicats opaques* existent, et chacune d'entre elles peut être renforcée en étant composée avec une autre technique d'*obfuscation*. Nous avons vu aussi qu'il existe

différentes méthodes de détection des *prédicats opaques*, chacune avec leurs avantages et leurs limites. Il n'existe pas de méthode universelle de détection. Dans la continuité de ce premier chapitre, nous nous limitons à l'analyse de programme contenant des *prédicats opaques* et interprétés par une analyse symbolique et concrète. Nous allons voir comment un *prédicat opaque* peut mettre en difficulté cette interprétation.

Chapitre 3

Déobfuscation symbolique et concrète

Sommaire

3.1 La déobfuscation automatique 37
3.2 Exécution dynamique symbolique 43
3.3 Mécanismes des solveurs *SMT* 58
3.4 Exemple d'un effet observable lors de l'analyse d'un prédicat opaque 72
3.5 Conclusion du chapitre 74

Résumé Nous présentons les processus automatiques dédiés à la réduction ou annulation des effets de l'obfuscation, appelés aussi méthode de déobfuscation. Nous essayons de comprendre comment lors d'une analyse symbolique et concrète, les prédicats opaques perturbent cette analyse. Nous abordons dans la section 3.1 les mécanismes de déobuscation lors de la reconstruction d'un *graphe de flot de contrôle*. Nous décrivons ensuite dans la section 3.2 le fonctionnement d'un moteur d'exécution symbolique dynamique utilisant un solveur *SMT*, dont nous détaillons le fonctionnement en section 3.3. Nous décrivons les difficultés rencontrées par ces outils lors de l'analyse de programmes contenant des prédicats opaques, dans la section 3.4.

3.1 La déobfuscation automatique

La déobfuscation s'exprime de manière classique comme la combinaison de deux actions liées. La première action détecte la présence d'une transformation de programme dont le but est d'ajouter du code difficile à interpréter (défini au moins dans la classe de complexité descriptive *NP-easy* [App02 ; Dun+12]), mais facile à exécuter. Autrement dit, un objet « sémantiquement significatif » est ajouté dans le programme et son interprétation directe ou sa modélisation est difficile [Yad+15]. La seconde action a pour but de retrouver un programme équivalent au

programme original ou de supprimer les effets de l'obfuscation. Cette action est la plus difficile à mettre en œuvre et est parfois impossible à réaliser [GR14 ; BDM17]. De nombreux travaux portent sur l'amélioration de ces deux actions. Une piste intéressante utilise certaines méthodes d'analyses sémantiques de codes comme un moyen de déobfusquer automatiquement un programme. Nous pouvons citer par exemple, les méthodes de *dévirtualisation* (*devirtualization*) [CLD11 ; Lia+17 ; SBP18] ou les méthodes de *synthèses* (*program synthesis*) [Bio+17 ; Bla+17]. Dans ces méthodes, un programme est interprété comme une collection ordonnée de transformations de deux ensembles de chaînes de caractères (les valeurs d'entrées et de sorties du programme) [Yad+15]. La figure 3.1 illustre cette représentation, dans laquelle `Entrées` et `Sorties` représentent deux ensembles de chaînes de caractères pouvant être acceptées ou retournées par le `Programme`. Nous pouvons interpréter aussi cette représentation comme une fonction acceptant et retournant des chaînes de caractères. L'objectif de ces méthodes est de réussir à vérifier ou extraire certaines propriétés contenues dans le programme, c'est-à-dire à juger de la présence ou non de ces propriétés. Dans ce contexte, la déobfuscation d'un programme consiste à vouloir rendre l'interprétation d'une propriété plus simple [CD05], sans nécessairement avoir la volonté de reconstruire le programme non obfusqué. Autrement dit, comme l'obfuscation compose le programme avec de nouvelles propriétés qui ont pour but de ralentir une ou plusieurs interprétations, la déobfuscation annihile ces propriétés. Nous souhaitons dans ce manuscrit pouvoir être en mesure de détecter une de ces propriétés.

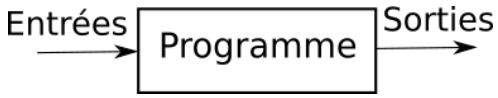


FIGURE 3.1 – Représentation d'un programme

Lorsque la quantité de programmes à analyser devient importante, l'automatisation est nécessaire. Nous introduisons maintenant le problème de l'analyse automatique d'un programme, puis nous définissons les trois mécanismes liés à l'analyse automatique : la capture de la structure, la simplification du code, et la reconstruction du *CFG*.

3.1.1 L'analyse statique automatisée

L'*analyse statique d'un programme* (*static program analysis*) permet de déterminer automatiquement si des propriétés d'exécution d'un programme défini existent durant toute l'exécution du programme, selon COUSOT et COUSOT [CC02]. Ce problème étant non décidable, l'interprétation abstraite de la sémantique du code est approximative. L'analyse statique pure est *conservative*

et *correcte* (*conservative and sound*) [1]. Ainsi, une interprétation concrète d'un programme permet d'obtenir une surapproximation de l'exécution de ce dernier [UDM05]. Les raisonnements imprécis et l'abstraction des éléments dépendant de l'environnement (bibliothèques, comportement du système d'exploitation) sont à l'origine de cette surapproximation [KK12]. Elle induit une imprécision additionnelle dans l'analyse, qui peut finir par produire une représentation du programme inutilisable pour la poursuite de l'analyse. Pour ces raisons, chaque analyseur possède sa propre représentation de l'information dans le but de proposer un équilibre entre approximation et efficacité des méthodes de raisonnement utilisées. Nous pouvons noter aussi que ces choix permettent de rendre l'analyse d'un programme réalisable dans des temps raisonnables. Nous définissons l'analyse statique comme la composition de trois actions : la capture de la sémantique, la simplification d'un objet, et la modélisation du graphe de flot de contrôle. Nous présentons maintenant ces trois actions.

3.1.1.1 Capturer la sémantique

La sémantique d'un programme est capturée à l'aide d'un langage du premier ordre. Les langages du premier ordre opposent au niveau syntaxique [2] deux grandes catégories d'objets : les constituants servant à identifier ou nommer des éléments du domaine (variables, symboles de constantes, termes), et les constituants servant à exprimer des propriétés ou des relations entre ces éléments (prédicats et formules). Le choix d'un langage du premier ordre permet de réaliser un compromis entre la précision de l'information manipulée et l'efficacité des algorithmes utilisés, par exemple lorsque le nombre de variables utilisées dans les formules est constant [Kop11]. Si la grammaire du langage du premier ordre est non ambiguë [3], et que cette extraction est précise, alors les outils d'analyse statique sont faciles à mettre en œuvre. En pratique, l'analyseur utilise au moins deux langages pour être efficace, et les traductions entre ces deux langages sont simples et rapides. Le premier langage utilisé par l'analyseur facilite la manipulation de l'information capturée. Le second langage permet la résolution de contraintes en faisant appel à un solveur. La combinaison de ces deux langages permet aussi la simplification d'un objet, comme nous allons le voir maintenant.

1. Les résultats de l'analyse statique peuvent être plus faibles que ceux recherchés, mais ils peuvent être généralisés à toutes les exécutions du programme.

2. L'ensemble des règles qui définissent les séquences correctes des éléments d'un langage de programmation.

3. Si chaque chaîne définie dans cette grammaire possède seulement une dérivation.

3.1.1.2 La simplification d'un objet

La *simplification* d'un objet est l'ensemble des actions permettant de redéfinir ce dernier, avec l'objectif de diminuer le temps nécessaire à son interprétation (pour réaliser des calculs ou des raisonnements). Une simplification est effectuée sous la condition que l'analyseur maîtrise l'ensemble des couples d'entrées/sorties de l'objet. En pratique, il n'existe pas de méthode universelle pour réaliser une simplification, car la méthode dépend du but (méthodes heuristiques, méthodes fonctionnelles, méthodes algébriques, méthodes propositionnelles, *etc.*), comme le montre l'exemple suivant.

Exemple 2:
Imaginons dans un programme l'existence d'une variable y définie par une fonction constante f (par exemple : $\{f(x) = 4; y = f(x);\}$). L'analyseur capture la définition de la variable y. Puis, il interprète cette définition avec une méthode lui permettant d'attester que la fonction est « constante ». Différentes solutions en pratique sont disponibles pour montrer que la variable y est constante, mais ces techniques vont dépendre de l'interprétation de y. Si y est vue comme une expression algébrique, l'analyseur va réécrire la définition de y en supprimant la définition de la fonction f, et en remplaçant le symbole $f(x)$ par la constante numérique (par exemple : $\{y = 4;\}$). La nouvelle définition de la variable y est devenue atomique (elle ne contient plus de sous-formule). Si y est vue comme une expression logique, comme le symbole de fonction f est égal au symbole constant 4, la propriété du symbole de fonction f est remplacée d'abord par la propriété d'un symbole constant, puis est remplacée par sa valeur numérique. Dans ce cas, l'effort de la machine pour simplifier la variable y est moindre. Ces deux méthodes permettent d'atteindre le même but. Cependant, la quantité d'opérations réalisée par ces deux méthodes ne sont pas les mêmes. Ainsi, le choix d'une méthode d'optimisation va dépendre des contraintes de calculabilité définie par l'environnement d'utilisation.

3.1.1.3 La modélisation du graphe de flot de contrôle

Les différents objets capturés pendant une analyse statique permettent d'extraire des propriétés du *CFG*. Ces propriétés permettent de créer une modélisation du programme exploitable par l'analyseur. Cette modélisation n'a pas pour but premier d'être retournée à l'utilisateur de l'analyseur. Obtenir ce modèle nécessite parfois la reconstruction (partielle ou totale) de quelques instructions, à partir des informations haut niveau connues (par exemple : les expressions conditionnelles et les boucles). En général, la difficulté principale vient du traitement des branchements indirects (par exemple, en langage assembleur *x86*, les instructions de sauts

comme `jmp eax`) ; en langage intermédiaire `LLVM-IR` [Lat08], les nœuds phi [4]. La cible de ces branchements est calculée normalement au moment de l'exécution du programme. En dehors des méthodes de réécriture ou de simplification des objets, lorsque l'analyseur rencontre un branchement indirect, un résolveur de contraintes permet de trouver une solution. Comme dit précédemment, le choix du langage et la méthode choisie pour résoudre ces contraintes influenceront la quantité de ressources nécessaires pour trouver une solution, si au moins une solution existe.

3.1.1.4 Analyse automatisée

Les techniques d'analyse automatisée constituent une utilisation spécifique de l'analyse statique, et peuvent être appliquées à la déobfuscation. Ces analyses n'autorisent pas, durant leurs exécutions, d'interactions avec les humains. Leur utilisation est intéressante lorsqu'une entité doit analyser un grand nombre de programmes régulièrement, par exemple pour l'analyse de *malwares*, ou la détection de codes malveillants dans des pages web. En pratique, Banescu *et al.* [Ban+16] font remarquer que ces analyses peuvent poursuivre deux objectifs différents :

- la modification d'un logiciel pour en changer le comportement (*software-tampering*)
- l'extraction d'un algorithme ou d'une donnée propriétaire

En pratique, le premier point s'effectue en plusieurs étapes : identification de la fonctionnalité, modification du processus de vérification d'intégrité du programme, et enfin modification de la fonctionnalité. Le second point peut être réalisé en simplifiant le *CFG* et les *flots de données* (*data-flow*) du programme. Le chapitre 4 de ce manuscrit illustre le premier point. Les chapitres 5 et 6 rentrent dans le cadre du second point. Nous présentons maintenant un exemple de déobfuscation.

3.1.2 Exemple de déobfuscation

La figure 3.3 illustre un exemple de déobfuscation automatique, adapté de Yadegari [Yad16]. Un petit programme en assembleur *x86* décrit une boucle réalisant des sauts indirects au travers d'éléments successifs d'un tableau T, en lecture seule. Ce programme est obfusqué, car il contient du code ayant pour seul but de rendre plus compliquée sa compréhension. Un analyseur statique interprète ce programme. Il commence l'interprétation au début du programme. Pour

4. Un nœud phi est une instruction utilisée dans la forme *statique à affectation unique* (*static single assignment* (*SSA*)), en début d'un *bloc de base* (*basic block*), permettant de sélectionner une valeur dépendante de la *structure de contrôle* (*control flow*)

chaque symbole rencontré, il définit un terme s'il n'existe pas, ou une expression à partir des termes déjà définis. Nous considérons dans cet exemple une gestion simplifiée de la figure 3.3. Nous décrivons maintenant le fonctionnement de l'analyseur. Les variables utilisées dans le programme sont définies comme des entiers. Le tableau est défini. La variable utilisée pour définir la position dans le tableau reçoit une propriété supplémentaire de pointeur de tableau, qui ajoute une opération arithmétique (listing 3.2). Chaque action du programme est ensuite parcourue. L'analyseur obtient une première modélisation du programme. Comme les valeurs de `ebx` sont connues (listing 3.3), l'analyseur peut simplifier le programme en propageant les différentes valeurs de `ebx` dans ce modèle. Le nouveau modèle correspond à la lecture des éléments du tableau (listing 3.4).

```
      mov ebx, 0
Lab:  mov eax, T[ebx]
      jmp [eax]
      add ebx, 4
      cmp ebx, 12
      jne Lab
```

Listing 3.1 – Code statique

T :	0x60000000
	0x62000000
	0x65000000

FIGURE 3.2 – Tableau T en mémoire (lecture seule)

```
def :: (T , tableau([1],[3], entier));
def :: (T[0] , /*expressions*/);
def :: (T[1] , /*expressions*/);
def :: (T[2] , /*expressions*/);
def :: (ebx , variable(entier));
def :: (eax , variable(entier));
ebx = 0;
T[ebx/4];
ebx = ebx + 4 ;
if(ebx < 12){
        T[ebx/4];
    ebx = ebx + 4 ;
        if(ebx < 12){
                T[ebx/4];
                ebx = ebx + 4 ;
        }}
```

Listing 3.2 – Premier modèle

```
def :: (T , tableau([1],[3], entier));
def :: (T[0] , /*expressions*/);
def :: (T[1] , /*expressions*/);
def :: (T[2] , /*expressions*/);
def :: (ebx , variable(entier));
def :: (eax , variable(entier));
ebx = 0;
T[0/4];
ebx = 0 + 4 ;
if(4 < 12){
        T[4/4];
    ebx = 4 + 4 ;
        if(8 < 12){
                T[8/4];
                ebx = 8 + 4 ;
        }}
```

Listing 3.3 – Second modèle

```
def :: (T , tableau([1],[3], entier));
def :: (T[0] , /*expressions*/);
def :: (T[1] , /*expressions*/);
def :: (T[2] , /*expressions*/);
T[0];
T[1];
T[2];
```

Listing 3.4 – Modèle final

FIGURE 3.3 – Exemple de déobfuscation basée sur [Yad16]

Bilan

Dans cette section nous venons de voir les principes de fonctionnement de l'analyse statique L'obfuscation est un moyen d'entraver l'efficacité cette analyse. Un moyen pour réduire les effets de l'obfuscation est d'adapter un analyseur statique. Dans la section suivante, nous présentons une

famille d'outils réalisant une analyse statique d'un programme basé sur l'exécution symbolique et concrète.

3.2 Exécution dynamique symbolique

Les outils d'*exécution dynamique symbolique* (*dynamic symbolic execution* (*DSE*)) permettent de prouver l'*accessibilité* (*reachability*) de certaines parties d'un code (tel que les chemins, les branches ou les instructions). Ces outils sont utilisés dans deux domaines distincts : *le test et l'analyse de programmes* (*program testing and analysis*) pour générer des tests automatiquement, et en *sécurité informatique* (*computer security*) principalement pour de la découverte de vulnérabilité. Ces outils s'appuient sur l'*exécution symbolique* (*symbolic execution*) et l'*exécution concrète* (*concrete execution*). L'objectif est de générer le plus rapidement possible les conditions d'utilisation d'un programme, c'est-à-dire de générer un ensemble de situations qui illustrent les différents comportements d'un programme. Les avantages d'une *exécution symbolique dynamique* sont les suivants, d'après David [Dav17] :

- le chemin parcouru est faisable en pratique. L'exécution du programme devient correcte (*sound*) ;
- la prochaine instruction exécutée après le calcul est toujours connue, cette propriété est intéressante au niveau binaire lorsque l'analyseur agit sur les sauts indirects ;
- toutes les boucles sont déroulées : cette propriété permet de garantir que chaque variable contenue dans un tour de boucle est utilisée uniquement pour un tour, ce qui facilite l'analyse statique.

Les moteurs d'*exécution dynamique symbolique* peuvent explorer les programmes de deux manières différentes : en fonctionnement *avant* (*forward*) ou en fonctionnement *arrière* (*backward*). Les outils d'*exécution dynamique symbolique* rencontrent plus de difficultés à analyser des programmes protégés à l'aide de prédicats opaques lors d'une analyse en *avant* que lors d'une analyse en *arrière*. C'est pourquoi, dans ce manuscrit, dans le but de réduire les effets des prédicats opaques utilisés dans le cadre de l'obfuscation de programmes, nous faisons le choix d'utiliser des outils d'*exécutions dynamiques symboliques* explorant les programmes en *avant*. Dans le reste de cette section, nous introduisons le fonctionnement d'un moteur d'*exécution symbolique dynamique*, les deux utilisations possibles du moteur pour réaliser des essais, et terminons par deux exemples d'outils existants.

3.2.1 Exécution symbolique et exécution concrète

3.2.1.1 Exécution symbolique

Lors d'une *exécution symbolique* (*symex* ou *symbolic execution*), un programme est interprété comme un ensemble de variables et de relations, défini dans une théorie. Chaque séquence de relations construite de cette manière permet de définir un chemin dans un programme. Plus précisément, l'analyse comportementale d'un programme basée sur l'exécution symbolique, appelée *analyse symbolique* (*symbolic analysis*), consiste à collecter des contraintes en parcourant différents chemins du *CFG*. Ces contraintes sont exprimées dans un langage logique, et résolues à l'aide d'un solveur. Lorsqu'un chemin est parcouru par l'*exécution symbolique*, une formule propositionnelle représentant l'ensemble des contraintes rencontrées est mise à jour de manière dynamique. Lorsqu'un nouveau chemin est rencontré, une nouvelle formule est créée, pour accueillir les contraintes rencontrées. L'ancienne formule n'est alors plus mise à jour. La résolution de la formule de l'ancien chemin permet de connaître les conditions d'accès à ce nouveau chemin. L'analyse d'un programme consiste en la résolution de l'ensemble des formules contenant les contraintes rencontrées. Cet ensemble est appelé la *représentation symbolique* (*symbolic representation*) du programme, et permet de définir différents modèles du programme, c'est-à-dire différents ensembles d'assignations des variables d'entrées d'un programme, permettant d'exprimer les différents comportements du programme. La partie du moteur capable de réaliser l'interprétation symbolique est appelée machine symbolique. Ce moteur maintient pour chaque chemin parcouru une formule logique décrivant les conditions satisfaites par les branches prises le long de ce chemin, et un magasin de mémoire symbolique qui relie les variables des expressions symboliques à des valeurs. Pour illustrer le fonctionnement de l'analyse symbolique, nous utiliserons le programme exemple TOY (figure 3.4). Ce programme réalise deux opérations arithmétiques, et propose de vérifier une condition.

l_0 : `int a, b;`
l_1 : `x := a` $\oplus$ `b;`
l_2 : `x := 2` $\times$ `x;`
l_3 : // assert x $>$ 15

FIGURE 3.4 – Programme exemple TOY

Représentation parfaite et imparfaite La représentation symbolique est dite *parfaite* [God12] lorsqu'elle permet de recouvrir l'ensemble des exécutions possibles du programme, ce qui correspond souvent à l'ensemble des chemins d'exécution du programme. Si un chemin n'est pas

couvert (ou *mal couvert* dans le sens où certaines relations ne sont pas traduites de manière équivalente par la représentation symbolique), alors la représentation est dite *imparfaite*. Cette représentation *imparfaite* existe aussi lorsque l'expressivité du langage symbolique ne permet pas d'exprimer toute la sémantique du programme. Sauf indications contraires, nous supposerons dans ce manuscrit que les outils utilisés réalisent une représentation *parfaite*. L'exemple 3 illustre un cas de représentation *imparfaite*.

Exemple 3:
Dans la figure 3.5, l'interprétation de l'expression de la ligne l_1 du programme TOY dans un langage ne capturant pas la relation $\oplus$ n'est pas équivalente à l'expression d'origine. La relation $\oplus$ est remplacée par une relation vide (représentée par le symbole $\square$).

$$\text{Interpretation}(\{\mathtt{x} := \mathtt{a} \oplus \mathtt{b}\}) \longrightarrow \{\mathtt{x} = \mathtt{a} \square \mathtt{b}\}$$

FIGURE 3.5 – Exemple de représentation *imparfaite* (la relation $\oplus$ n'est pas conservée)

3.2.1.2 Les défis d'un moteur symbolique

Un moteur symbolique transforme un programme en représentation symbolique. Plus précisément, les entrées du programme sont remplacées par des symboles abstraits, et chaque instruction ou opération contenue dans le programme est remplacée par une formule abstraite équivalente. Le moteur simule un ensemble d'exécutions d'un programme, en assurant le suivi et la propagation des entrées symboliques dans les formules abstraites. Autrement dit, il réalise une interprétation du programme. Cette interprétation se matérialise en créant un *arbre symbolique* (*symbolic tree*). Cet arbre possède des *étiquettes* (*labels*) qui correspondent aux contraintes rencontrées, et permettent de créer les *prédicats de chemin* (*path predicate*). Les conditions de branches et les assignations rencontrées durant le parcours d'un chemin d'exécution permettent de définir un *prédicat de chemin*. En fonction des buts par l'analyse d'un programme, différents choix et compromis doivent être faits par le moteur symbolique afin de rendre l'exécution symbolique efficace. Ces choix concernent l'efficacité de la machine symbolique, les contraintes de ressources, et la représentation de la mémoire.

Efficacité de la machine symbolique L'efficacité de la machine symbolique repose sur les trois composants suivants :

- une stratégie de planification d'exploration des chemins, permettant de parcourir un chemin approprié pour réaliser l'exploration souhaitée,

- un algorithme de construction des prédicats de chemin,
- une méthode de vérification de la faisabilité (*satisfiability checking*) des chemins, utilisant un solveur de contraintes propositionnelles.

Suivant l'outil d'*exécution symbolique dynamique*, le choix des stratégies de planification d'exploration des chemins peut être fixé ou non durant l'analyse. Les stratégies de planification d'exploration des chemins les plus courants sont les suivantes : *depth first search* qui minimise la consommation de ressources, *coverage optimized search* qui a pour but de recouvrir le maximum d'instructions exécutables (couverture maximale du programme) et *random path selection* qui espère compenser le manque de ressources avec une exploration aléatoire. L'exemple 4 illustre l'exploration du programme TOY (figure 3.4), et les différents comportement de ce programme.

La mise à jour des *prédicats de chemin* passe par le choix d'une décision parmi ces trois cas possible :

- *concrétisation* : calcul d'une valeur afin de sous-approximer une partie de l'exécution lorsque celle-ci est manquante (par exemple les appels système) ou trop coûteuse en raisonnement symbolique (par exemple les fonctions de hachage).
- *symbolisation* : remplace certaines définitions de variables par une variable symbolique sans contrainte, permettant de généraliser certaines étapes du programme.
- *propagation* : réalise un calcul symbolique du prédicat de chemin, sans réaliser de sur approximation du chemin.

Exemple 4:
Dans notre programme TOY, il n'existe qu'un seul chemin d'exécution, représenté par les lignes l_0-l_1-l_2-l_3. La stratégie d'exploration interprète la sémantique de chaque ligne pour mettre à jour le magasin symbolique (figure 3.6). Ce magasin associe les variables du programme pour toutes les expressions avec des valeurs symboliques α_i ou concrètes.

$$\{\mathtt{a} \mapsto \alpha_a \ , \mathtt{b} \mapsto \alpha_b, \mathtt{x} \mapsto (2 \times (\alpha_a \oplus \alpha_b))\}$$

FIGURE 3.6 – Magasin symbolique du programme TOY

La condition devant être résolue pour fournir un contre-exemple est :

$$(2 \times (\alpha_a \oplus \alpha_b)) \leq 15$$

FIGURE 3.7 – Condition du programme TOY

Le type des variables `a` et `b` étant des entiers non-signés, les solutions de cette condition peuvent être trouvées à l'aide d'un solveur *SMT*. Les modèles de cette condition sont :

$$\{\{\alpha_a = 0, \alpha_b = 7\}, \{\alpha_a = 1, \alpha_b = 6\},$$
$$\{\alpha_a = 2, \alpha_b = 5\}, \{\alpha_a = 3, \alpha_b = 4\}, \{\alpha_a = 4, \alpha_b = 3\},$$
$$\{\alpha_a = 5, \alpha_b = 2\}, \{\alpha_a = 6, \alpha_b = 1\}, \{\alpha_a = 7, \alpha_b = 0\}\}$$

FIGURE 3.8 – Modèles concrets de la condition du programme TOY

Comme l'analyse statique parfaite d'un programme est impossible, la représentation symbolique réalisée par un prédicat est un compromis entre la précision du prédicat à exprimer la structure du chemin d'exécution (l'ensemble des instructions suivies durant une exécution) et les vraies valeurs que ce chemin peut prendre. Autrement dit, chaque prédicat obtenu est un équilibre entre être le plus *correct* (définition 3.1) et être le plus *complet* (définition 3.2), pour éviter que le prédicat soit *décohérant* (définition 3.3). Par exemple, un prédicat est *décohérant* lorsque le langage utilisé ne permet pas de capturer certaines opérations définies dans le programme.

Définition 3.1 *Un prédicat est complet si chaque entrée couvrant le chemin d'exécution regardé est une solution du prédicat.*

Définition 3.2 *Un prédicat est correct si chaque solution de ce prédicat permet de couvrir le chemin d'exécution regardé.*

Définition 3.3 *Un prédicat est décohérant si la représentation symbolique est imparfaite, c'est-à-dire que la représentation symbolique ne représente pas correctement le chemin d'exécution regardé.*

Exemple 5:
Les conditions définies dans le Tableau 3.1 représentent le prédicat de chemin du programme exemple TOY. φ_1 est le prédicat de chemin original du programme TOY. φ_2 est le prédicat de chemin symbolique, c'est-à-dire qu'une variable est remplacée par une variable *fraîche* (*fresh*) symbolique (une variable sans contraintes). φ_3 est le prédicat de chemin concret, c'est-à-dire une valeur est assignée à une variable d'une des expressions (ici la variable `a` reçoit la valeur 5). Cette valeur peut être choisie de manière aléatoire, ou bien être déduite d'un ensemble de conditions existantes avant ce programme. Le prédicat φ_1 est *correct* et *complet*, car toutes

les entrées couvrant le chemin d'exécution sont solutions, et inversement. Le prédicat φ_2 est *correct*, mais incomplet, car toutes les solutions trouvées recouvrent le chemin d'exécution, et pas l'inverse. Le prédicat φ_3 est *complet* et incorrect, car toutes les entrées trouvées couvrant le chemin d'exécution sont solutions, mais toutes les solutions ne couvrent pas le chemin d'exécution.

prédicat de chemin φ_1	prédicat symbolique φ_2	prédicat concret φ_3
$\mathtt{x}_1 = \mathtt{a} \oplus \mathtt{b}$ $\wedge\ \mathtt{x}_2 = 2 \times \mathtt{x}_1$ $\wedge\ \mathtt{x}_2 > 15$	$\mathtt{x}_1 = fresh$ $\wedge\ \mathtt{x}_2 = 2 \times \mathtt{x}_1$ $\wedge\ \mathtt{x}_2 > 15$	$\mathtt{a} = 5$ $\mathtt{x}_1 = 5 \oplus \mathtt{b}$ $\wedge\ \mathtt{x}_2 = 2 \times \mathtt{x}_1$ $\wedge\ \mathtt{x}_2 > 15$

TABLE 3.1 – Exemples de prédicats de chemin, de symbolisation et de concrétisation, du programme exemple TOY

Contraintes de ressources Les ressources d'une machine sont exprimées en quantité de temps et quantité de mémoire. Nous nous limitons aux contraintes de ressources en lien avec l'obfuscation. D'après l'étude de BANESCU *et al.* [Ban+16], ces contraintes génèrent principalement des prédicats décohérants. Les conséquences sont de trois types :

1. l'explosion combinatoire du nombre de chemins (*path explosion*) de l'arbre symbolique,
2. la difficulté de résolution de certaines contraintes pour le moteur d'exécution symbolique,
3. la difficulté de ne pas explorer certains chemins inutiles.

Pour limiter le premier problème, l'analyseur utilise une base de connaissances. Par exemple, un ensemble de contraintes peuvent être fixées sur les entrées symboliques en fonction de l'environnement d'exécution (taille des contraintes connues, préfixes, valeurs concrètes, constantes, *etc.*). La résolution de contraintes difficiles pour le moteur d'exécution symbolique ne correspond pas toujours à un problème difficile à résoudre. Les moteurs d'exécution symbolique sont conçus pour résoudre une collection de problèmes de manière efficace. Parfois, comme le montrent les prédicats opaques, des contraintes peuvent ne pas contenir un problème difficile, mais peuvent cependant mettre du temps à être résolues. Ce qui les rend difficiles pour le moteur d'exécution symbolique. L'exemple 6 illustre un renforcement du programme TOY face à une analyse symbolique.

Exemple 6:
Si dans le programme exemple TOY (figure 3.4), l'expression de la ligne l_2 est composée avec une fonction de hachage, et devient $\{l'_2 : \texttt{x} := \texttt{hash}(2 \times \texttt{x})\}$, alors le magasin symbolique obtenu pour le programme est :

$$\{\texttt{a} \mapsto \alpha_a \ , \texttt{b} \mapsto \alpha_b, \texttt{x} \mapsto (\texttt{hash}(2 \times (\alpha_a \oplus \alpha_b)))\}$$

FIGURE 3.9 – Nouveau magasin symbolique du programme TOY

La nouvelle condition est :

$$\texttt{hash}(2 \times (\alpha_a \oplus \alpha_b)) \leq 15$$

FIGURE 3.10 – Nouvelle condition du programme TOY

La résolution de cette condition suppose que le moteur symbolique possède une définition de la fonction de hachage `hash`. Résoudre cette condition symboliquement revient à vouloir inverser la fonction `hash`. Or, si cette fonction est cryptographique [a], inverser cette fonction en un temps raisonnable n'est pas envisageable, voir impossible. C'est une limite de l'analyse symbolique.

a. Une fonction de hachage est cryptographique si elle possède les propriétés suivantes : résistance à la préimage, résistance à la seconde préimage, résistance aux collisions.

Représentation de la mémoire L'équilibre entre la représentation de l'information et l'efficacité des algorithmes oblige un choix sur la représentation de la mémoire, et plus particulièrement sur les lectures et écritures symboliques. Trois approches sont possibles :

1. *entièrement symbolique* : toutes les représentations peuvent être considérées ;
2. *entièrement concrète* : une seule représentation est considérée ;
3. *partiellement symbolique et partiellement concrète* : les écritures de données sont concrètes, les chargements des données sont parfois concrétisés lorsque la résolution est difficile.

La représentation *entièrement symbolique* : de la mémoire permet d'obtenir N états pour chaque variable, mais elle est difficilement utilisable sur tout un programme entier. La représentation *entièrement concrète* permet d'avoir un seul état pour chaque variable, sans permettre d'être précis. La représentation *partielle* permet d'obtenir un ensemble de K états pouvant être inexacts.

3.2.1.3 Exécution concrète

Dans une exécution concrète, des instructions d'un programme sont exécutées pour une entrée spécifique et explorent un seul chemin du flot de contrôle. L'exécution concrète est utilisée pour obtenir différentes informations : obtenir les intervalles des adresses en mémoires de chaque pointeur contenu dans un programme, obtenir le résultat d'une fonction définie dans une bibliothèque. Les méthodes d'exécution varient en fonction des moteurs utilisés. Par exemple, l'exécution peut se faire instruction par instruction directement sur la machine hôte, pour récupérer les valeurs des mémoires et registres. Un solveur *SMT* peut aussi réaliser une exécution concrète : si un prédicat de chemin est satisfiable, alors un modèle représente une exécution concrète de ce chemin. Les exécutions concrètes permettent d'obtenir une sous-approximation de la propriété analysée.

3.2.2 Deux approches différentes

Un moteur d'exécution symbolique dynamique a pour but de retourner le plus rapidement possible des exemples reflétant le comportement d'un programme. Sa stratégie d'exploration vise à reproduire tous les états d'exécution du programme, en s'appuyant à la fois sur le moteur symbolique et sur des états d'exécution concrets. Il existe deux approches : les *essais concoliques* [5] (*concolic testing*) [GLM12], et les *essais générés par l'exécution* (*execution-generated testing*) [CE05]. Elles diffèrent dans leur façon de fonctionner. Les *essais concoliques* sont dirigés par l'exécution concrète et explorent chaque chemin du programme un par un. Ce qui n'est pas le cas pour les *essais générés par l'exécution* qui sont dirigés par l'exécution symbolique et l'exécution *bifurque* (*fork*) chaque fois que l'analyse rencontre une nouvelle branche dans le programme.

3.2.2.1 Essais concoliques

Les *essais concoliques* sont limités en temps et respectent les étapes suivant :

1. un programme est exécuté pour une entrée fixée ;
2. la condition d'exécution du chemin (*path condition*) est collectée durant cette exécution, à l'aide d'un *état de la mémoire symbolique* (*symbolic memory state*) ;
3. la condition de chemin avec la négation d'une branche est résolue à l'aide d'un solveur *SMT* pour fabriquer la prochaine entrée.

5. Mot-valise des mots *concrete* et *symbolic*

L'algorithme 1 représente un algorithme d'*essais concoliques*, présenté initialement par CHA *et al.* [Cha+19]. Nous décrivons maintenant le fonctionnement de cet algorithme. L'algorithme accepte en entrée un programme P, un vecteur d'entrée v_0, et le nombre N d'exécutions qu'il peut réaliser. L'algorithme met à jour régulièrement un arbre d'exécution T, contenant la liste des conditions de chemins précédemment explorées pour ce programme. L'arbre d'exécution T et le vecteur d'entrée v sont initialement vides. Ligne 4, la fonction `RunProgram` réalise une *exécution concrète* du programme P pour l'entrée v, permettant d'obtenir le chemin d'exécution courant Φ_m. Le chemin de condition est ajouté à l'arbre T (ligne 5). Lignes 6-8, l'algorithme choisit une branche et exprime sa négation. La fonction `Choose` correspond à la fonction de recherche heuristique et choisit un chemin de condition Φ depuis T, puis sélectionne une branche ϕ_i, définie dans Φ. Pour une branche ϕ_i fixée, l'algorithme va fabriquer une nouvelle condition de chemin $\{\Phi' = \bigwedge_{j<i} \phi_j \wedge \neg\phi_i\}$. Si Φ' est satisfiable, alors la prochaine entrée peut être générée à l'aide du solveur *SMT*. Sinon, si Φ' n'est pas satisfiable, l'algorithme va essayer la négation d'une autre branche jusqu'à ce que la satisfiabilité d'une condition de chemin soit trouvée. Ce processus est répété N fois, et le nombre final de chemins couverts est retourné.

Algorithme 1 : Essais concoliques [Cha+19]

Input : Program P, Initial input v_0, Integer N
Output : The number of branches covered

```
1  T ← ⟨⟩
2  v ← v_0
3  for m = 1 to N do
4  |   Φ_m ← RunProgram(P,v)
5  |   T ← T · Φ_m
6  |   do
7  |   |   (Φ,φ_i) ← Choose(T)                    // (Φ = φ_1 ∧ ··· ∧ φ_n)
8  |   while ¬SAT(⋀_{j<i} φ_j ∧ ¬φ_i)
9  |   v ← Model(⋀_{j<i} φ_j ∧ ¬φ_i)
10 return |Coverage(T)|
```

3.2.2.2 Essais générés par l'exécution

Les *essais générés par l'exécution* fonctionnent de la même manière que l'*exécution symbolique* et utilisent l'*exécution concrète* lorsque la résolution d'une condition de branchement n'est pas possible, ou bien lorsque l'instruction courante ne peut supporter une valeur symbolique. La satisfiabilité des expressions est déterminée à l'aide d'un solveur *SMT*. Cette méthode d'analyse

comporte plusieurs avantages comme le fait qu'elle facilite la modélisation des interactions du programme avec son environnement (par exemple : les appels systèmes, les bibliothèques non instrumentées). Le moteur extrait seulement la partie du code intéressante, c'est-à-dire que le moteur n'a pas besoin de transformer tout le code de manière symbolique. Cela permet d'obtenir un nombre réduit de *faux positifs* [6]. Plus précisément, les valeurs concrètes fabriquées durant cette analyse sont stockées dans une table, appelée *magasin concret* (*concrete store*), et l'ensemble des contraintes sont stockées dans une autre table, appelée *contraintes de chemins* (*path constraints*). Un *état d'exécution* (*execution state*) définit le programme à un point particulier d'une exécution concrète, par l'intermédiaire d'une ou plusieurs instructions. L'ordre des éléments stockés dans ces tables correspond à l'ordre des informations rencontrées durant l'exploration du programme depuis son point d'entrée. Ces deux tableaux constituent le *magasin symbolique* (*symbolic store*), et donnent une modélisation du programme. Les magasins sont périodiquement simplifiés à l'aide des techniques de *conditionnement de programme* (*program conditioning techniques*) [Sch+16]. Cette simplification dépend de deux conditions, d'après les travaux de Danicic *et al.* [Dan+05] :

- la précision de l'exécution symbolique au niveau de la transmission des informations dans le chemin parcouru (par exemple : les informations échangées entre deux états symboliques),
- la précision du solveur de contrainte propositionnelle sur la résolution des contraintes

L'algorithme 2 représente un algorithme d'*essais générés par l'exécution*, présenté initialement par Cha *et al.* [Cha+19]. Nous présentons maintenant cet algorithme. L'algorithme prend en entrée un programme P, et un nombre N d'étapes (ou de temps). L'ensemble des états explorés $States$ est initialisé avec l'état initial $(instr_0, S_0, true)$. L'ensemble des cas d'essais T est l'ensemble vide. L'exemple 7 ci-après illustre le début de l'analyse d'un programme.

Exemple 7:
Si nous considérons le programme suivant :

```
1 void bar(int x){
2         if(x == 100)
3                 assert(x == 0 && "error msg");
4         }
```

Listing 3.5 – Exemple de programme

6. Ici, un *faux positif* est un prédicat de chemin incorrect. Cette situation apparaît par manque d'information, ou lorsque la décomposition de code est difficile.

L'état initial du programme est :

$$(\texttt{if(x} == \texttt{100)}, [x \rightarrow \alpha], true) \tag{3.1}$$

Puis, à la ligne 4, la fonction `Choose` sélectionne un état à explorer depuis l'ensemble $States$. À la ligne 6, la fonction `ExecuteInstruction` réalise une exécution concrète de l'instruction $instr$ de l'état sélectionné. Plusieurs types d'instructions peuvent être exécutées (par exemple : assignement, assertion, conditionnelle, halt, *etc.*). Nous considérons le cas d'une instruction conditionnelle et d'arrêt. Si $instr'$ est une déclaration conditionnelle (ligne 7), l'algorithme va vérifier si les nouvelles conditions de chemins pour les branches TRUE (ligne 8) ou FALSE (ligne 10) sont satisfiables. Si c'est le cas, l'algorithme bifurque en deux états : $(s_1, S', \Phi \wedge e)$ et $(s_2, S', \Phi \wedge \neg e)$.

Exemple 7: (suite)
Si nous reprenons notre exemple, l'état initial se sépare en deux états :

$$\left\{\begin{array}{r} state_1 = (\texttt{assert((x} == \texttt{0\&\&"errormsg"}), [x \rightarrow \alpha], \alpha = 100) \\ state_2 = (halt, [x \rightarrow \alpha], \alpha \neq 100) \end{array}\right\}$$

Quand $instr'$ est une déclaration d'arrêt, l'algorithme fabrique un modèle de Φ, et l'ajoute dans l'ensemble T. L'algorithme répète ce processus pour toutes les instructions jusqu'à ce que le nombre d'étapes maximum N soit atteint ou bien que l'ensemble $States$ soit vide (ligne 14). Les lignes 15-16 fabriquent des cas d'exemples en utilisant les conditions de chemin des états $States$, où les instructions de chaque état ne sont pas terminées. Enfin, en utilisant ces cas d'exemples T, l'algorithme retourne le nombre de branches couvertes (ligne 17).

3.2.3 Exemples de moteurs d'exécution symbolique dynamique

Nous trouvons dans la littérature deux moteurs d'*exécution symbolique dynamique* permettant d'analyser des programmes obfusqués : `KLEE` [7] [CDE+08] et `angr` [8] [Sho+16]. Ces moteurs sont considérés comme représentatifs de l'état de l'art de l'exécution symbolique dynamique. Nous les avons sélectionnés pour mener à bien nos expériences . Ces deux moteurs n'ont pas été développés dans le même but. `KLEE` est un outil rapide et entièrement automatique. `angr` réalise une analyse semi-automatique efficace (l'utilisateur peut modifier ou combiner différentes stratégies à sa guise). Ils n'utilisent pas la même représentation intermédiaire (`KLEE` utilise le langage

7. `https://klee.github.io/`
8. `https://angr.io/`

Algorithme 2 : Essais générés par exécutions [Cha+19]

Input : Program P, Integer N
Output : The number of branches covered

$States \leftarrow \{(instr_0, S_0, true)\}$
$T \leftarrow \emptyset$
do
  $(instr, S, \Phi) \leftarrow$ `Choose` $(States)$
  $States \leftarrow States \setminus \{(instr, S, \Phi)\}$
  $(instr', S', \Phi) \leftarrow$ `ExecuteInstruction` $(\{instr, S, \Phi\})$
  if $instr' =$*(if(e) then* s_1 *else* s_2*)* **then**
    if *SAT(*$\Phi \wedge e$*)* **then**
      $States \leftarrow States \cup \{(s_1, S', \Phi \wedge e)\}$
    if *SAT(*$\Phi \wedge \neg e$*)* **then**
      $States \leftarrow States \cup \{(s_2, S', \Phi \wedge \neg e)\}$
  else if $instr' =$ *halt* **then**
    $T \leftarrow T \cup$ model(Φ)
while *(*$N == 0$*)*$\vee$*(*$States == \emptyset$*)*
forall *(*$instr, S, \Phi$*)* $\in States$ **do**
  $T \leftarrow T \cup$ model(Φ)
return |*Coverage(T)*|

LLVM-IR [9] ; angr utilise le langage VEX-IR [10]), ni les mêmes heuristiques pour les stratégies de recherche. Nous allons présenter maintenant ces deux moteurs plus en détail.

3.2.3.1 KLEE

KLEE est un moteur d'exécution symbolique dynamique, développé en C++. Il s'appuie sur le langage intermédiaire LLVM-IR et est compatible avec l'utilisation de plusieurs solveurs *SMT* (STP, Z3, CVC4, Yices, Boolector), au travers de requêtes définies dans la théorie QF-AUFBV au format *SMT-LIB* [BST+10]. Sa stratégie principale s'appuie sur des *essais générés par l'exécution* (*execution-generated testing*) [CE05]. Sa rapidité d'exécution repose sur sa capacité à utiliser une méthode avancée d'élagage de trajectoires (*path merging*), de sa possibilité d'utiliser la bibliothèque standard C μClibc [11] pour simplifier les interactions entre le programme et son environnement, et enfin de ses différents caches lui permettant de limiter les interactions avec le solveur *SMT*. KLEE est utilisé dans des contextes très variés [Lap18 ; Cha+19]. En sécurité informatique, et plus particulièrement dans le domaine de la protection des programmes, KLEE est considéré comme l'attaquant le plus efficace possible pour des programmes obfusqués [Oll+19a].

Nous décrivons maintenant brièvement le fonctionnement de ce moteur. KLEE (figure 3.11) instrumente automatiquement la représentation intermédiaire d'un programme au format LLVM-IR, pour mettre en place des actions. Ces actions ont pour but de faciliter la capture du comportement du programme. Par exemple, si un programme contient un branchement conditionnel, KLEE ajoute une action de *supposition* (*assume*) permettant de vérifier cette condition. Ces actions permettent à KLEE de déduire le comportement du programme, sous certaines hypothèses, c'est-à-dire de définir, propager et résoudre des ensembles de conditions permettant le bon fonctionnement du programme. Pour être efficace en pratique, chaque état symbolique fabriqué par KLEE contient l'ensemble des conditions ayant permis d'atteindre cet état, comme le montre la figure 3.13. Ainsi, lorsqu'un ensemble de conditions ne peuvent être déduites des hypothèses courantes, KLEE construit une requête de satisfiabilité dans une théorie du premier ordre, interprétable par un solveur *SMT*. Si la requête est satisfiable, alors KLEE demande un modèle au solveur, puis garde en mémoire les résultats de cette requête, et utilise ces résultats pour le propager dans l'analyse. KLEE utilise une *pile du solveur* (*solver stack*), décrite dans la figure 3.12.

Si la requête n'est pas satisfiable, par une *méthode de va-et-vient* (*back and forth method*),

9. https://llvm.org/docs/LangRef.html
10. https://docs.angr.io/advanced-topics/ir
11. https://uclibc.org/

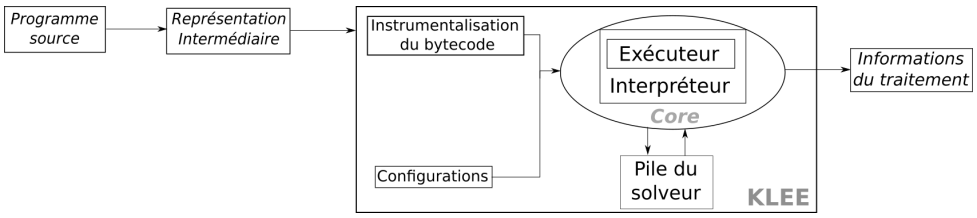


FIGURE 3.11 – Fonctionnement simplifié de KLEE

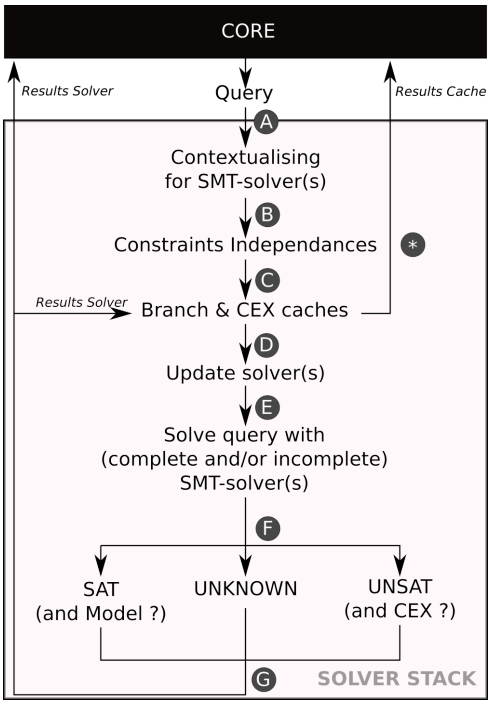


FIGURE 3.12 – Pile de solveur de KLEE

KLEE utilise la justification de la requête non satisfaite pour construire une nouvelle requête, et répète cette opération jusqu'à trouver une requête satisfiable ou jusqu'à ce qu'il n'y ait plus de justification. Dans ce dernier cas, KLEE arrête d'explorer le chemin en cours, sinon l'analyse continue.

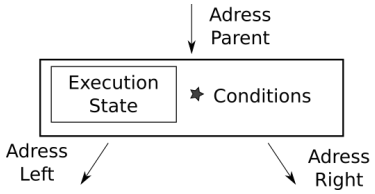


FIGURE 3.13 – Exemple d'état symbolique dans KLEE

3.2.3.2 angr

angr (figure 3.14) est un ensemble d'outils, développés en Python, permettant d'analyser des binaires à l'aide du langage intermédiaire VEX-IR (basé sur Valgrind [12]), utilisé parfois pour analyser des codes obfusqués. Il permet de réaliser des analyses statiques automatiques et semi-automatiques [13]. angr contient un outil d'exécution symbolique dynamique, effectuant des essais concoliques. La technique d'exécution symbolique utilisée par angr est basée sur les travaux de MAYHEM [Cha+12], permettant d'utiliser le modèle de mémoire basé sur un index et des stratégies donnant de l'importance à certains chemins. L'architecture d'angr utilise différents sous-projets : CLE (chargeur de binaire), archinfo (support pour la gestion de différentes architectures), PyVex (traduction du code assembleur en langage intermédiaire), et Claripy (l'interface entre le moteur d'exécution symbolique et le solveur *SMT*). angr se base sur une version légèrement modifiée du solveur *SMT* Z3. Cette version du solveur conserve les mêmes procédures de décision que la version originale. La communication avec ce solveur se fait au moyen de requêtes, définies dans la théorie QF-AUFBV au format *SMT-LIB* [BST+10].

Bilan

Nous venons de voir les principes de fonctionnement des moteurs d'exécution symbolique dynamique. Ces outils permettent de répondre à certaines faiblesses de l'analyse statique à l'aide d'un solveur *SMT*, et d'une stratégie d'exploration. Leur but est de créer le plus rapidement possible un ensemble de contraintes afin de connaître les conditions d'entrées permettant d'obtenir

12. https://valgrind.org/
13. L'utilisateur peut être appelé pour quelques choix de décisions de l'analyse.

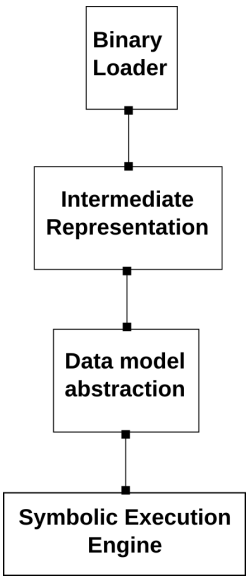


FIGURE 3.14 – Fonctionnement simplifié de `Angr`

les différents comportements d'un programme. `KLEE` ou `angr` sont des exemples représentatifs de ces outils. L'efficacité de l'analyse dépend en grande partie de l'utilisation des solveurs *SMT*. Nous proposons de voir dans la prochaine section les mécanismes de fonctionnement de ces solveurs.

3.3 Mécanismes des solveurs *SMT*

L'utilisation d'une machine pour effectuer tous les raisonnements possibles automatiquement est un vieux rêve de l'humanité. Nous avons vu dans ce manuscrit que l'analyse statique d'un programme nécessite des prises de décisions. L'utilisation des *solveurs de satisfiabilité modulo théorie*, ou plus simplement des solveurs *SMT*, permet de répondre à ces interrogations [DHK12]. La résolution d'un *problème de décision* à l'aide d'un automate fut proposée pour la première fois par BÜCHI [Büc60] et ELGOT [Elg61], suivis des travaux de HODGSON [Hod82] qui proposèrent d'améliorer cette technique à l'aide d'une théorie décidable. Les solveurs *SMT* sont développés pour résoudre une ou plusieurs instances du problème *SMT*, c'est-à-dire une adaptation du problème SAT à des logiques plus expressives basées sur des théories du premier ordre. Nous rappelons que le problème SAT est la restriction au domaine booléen d'un *problème de satisfaction de contraintes* (*constraint satisfaction problem* (*CSP*)) consistant à déterminer l'existence d'une affectation aux variables d'une formule Booléenne permettant de l'évaluer à VRAI. Par exemple, pour une formule $\phi = (a \Leftrightarrow \neg b) \wedge (\neg a \Rightarrow b)$, une affectation satisfaisante

de ϕ est $\{a \mapsto 0, b \mapsto 1\}$.
Le problème *SMT* fait apparaître un défi technique entre la *faisabilité* (*tractability*) et l'*expressivité* (*expressivity*) d'un raisonnement automatique [LB84]. Le langage d'entrée standard pour les solveurs *SMT* fut proposé par l'initiative *SMT-LIB* [BST+10], avec un large choix d'*étalons* (*benchmarks*). Cette initiative fut suivie de la mise en place d'une compétition annuelle, appelée *SMT Competition* (*SMT-COMP*) [19a], pour alimenter les *étalons*. Le standard *SMT-LIB* et les compétitions permettent l'amélioration de la visibilité des solveurs *SMT* en informatique et au-delà. Pour améliorer cette base de données, nous avons soumis à la compétition *SMT-COMP* un ensemble de requêtes *SMT* contenant des prédicats opaques, décrites dans le chapitre 7.

3.3.1 Du solveur SAT au solveur *SMT*

Les progrès récents des solveurs *SAT* [Mos+01 ; ES03 ; BHM09] permettent de résoudre une large classe de problèmes réalistes. Ces solveurs fonctionnent sur le concept de l'apprentissage par des clauses (*clause-learning*), et sont appelés solveurs à *apprentissage des clauses par conflits* (*Conflict-Driven-Clause-Learning (CDCL)*). Ces solveurs sont une amélioration des solveurs *SAT* basés sur l'algorithme de *Davis-Logemann-Loveland* (*DPLL*). De manière très simplifiée, l'objectif d'un solveur *CDCL* est de trouver un modèle à une formule propositionnelle, c'est-à-dire un ensemble d'assignations permettant de satisfaire cette formule.

La résolution de la formule s'effectue par un ensemble de décision réalisé par la propagation d'informations booléennes le long des différentes relations de la formule, et par des résolutions unitaires permettant de trouver de nouvelles implications. Lorsqu'un conflit apparaît entre deux littéraux, deux solutions sont possibles : soit aucune décision ne peut être prise et la résolution s'arrête (c'est-à-dire que la formule n'est pas satisfiable), soit une *clause de conflit* (*conflict clause*) peut être trouvée.
Cette clause permet à l'algorithme *CDCL* de revenir sur un ou plusieurs niveaux de décisions précédents, d'ajouter cette clause à la base de connaissances et de redémarrer la résolution de la formule propositionnelle. Pour accélérer la résolution, le solveur *SAT* peut être utilisé en mode incrémental par la mise en place d'une boucle de rétroaction logique. Dans ce cas, en début de résolution, de nouvelles hypothèses (littéraux) sont ajoutées à la formule et une variable *fraîche* (*fresh*) (variable symbolique sans contraintes) est ajoutée à chaque clause. Cette variable *fraîche* est utilisée comme une mémoire autorisant l'ajout ou la suppression de variables ou de clauses.

Les solveurs *CDCL* étant des outils complexes avec de nombreuses options, il existe différentes manières de modéliser leur fonctionnement. Nous proposons ici deux modélisations différentes de ce solveur. La première modélisation nous permet de comprendre les grandes

lignes de son fonctionnement. La seconde en est une abstraction, afin d'en avoir une utilisation simple dans ce manuscrit. Nous commençons avec une première modélisation (algorithme 3) proposée par PIPATSRISAWAT et DARWICHE [PD10]. Dans ce modèle, nous pouvons voir les deux piliers d'un solveur SAT : le *moteur de prise de décision* (*decision-making engine*) et le *moteur d'apprentissage des clauses* (*clause-learning engine*). Le premier moteur permet de trouver une assignation satisfiable des variables. Le second moteur permet de renforcer la résolution unitaire.

Algorithme 3 : Un pseudo-code des solveurs SAT modernes [PD10]

1 Prendre des décisions et effectuer des résolutions unitaires (*unit resolution*) jusqu'à ce qu'une solution ou un conflit soit trouvé. `// decision-making engine`
2 Si une solution est trouvée, retourner SAT.
3 Si un conflit est trouvé, retourner UNSAT s'il n'y a pas de décision.
4 Dans les autres cas, établir une clause de conflit, sous certaines conditions, ajouter la clause de conflit à la base de connaissances, et recommencer à l'étape 1.
`// clause-learning engine`

Pour que nous puissions utiliser ce solveur (et ses extensions) facilement dans ce manuscrit, nous utilisons un modèle abstrait du solveur *CDCL*, représenté sous forme de règles [Tin10]. Les règles sont représentées au format *d'une affectation gardée* (*guarded assignment form*) [14] [KG07]. Les symboles utilisés sont définis de la manière suivante. M est une séquence de littéraux. $\bullet$ (*point de décision*) représente une assignation de vérité partielle. C est une valeur contenant soit la valeur no, soit une clause de conflit. F est un ensemble de clauses représentant une formule au format *CNF* [15]. Lit(X) représente l'ensemble des littéraux de la variable X. La machine à état est définie de la manière suivante :

- Les variables du système sont : M, F, C
- Un état est défini par $\langle \mathsf{M},\mathsf{F},\mathsf{C} \rangle$ ou *fail*
- L'état initial est $\langle (), F_0, \mathsf{no} \rangle$, où F_0 est un ensemble de clauses devant être satisfaites
- L'état final est défini par :
  - *fail* si F_0 n'est pas satisfait,
  - par l'état $\langle \mathsf{M},\mathsf{G},\mathsf{no} \rangle$ sinon, où G est équivalent à F_0 et M satisfait G

14. Par exemple, la règle $\textbf{Rule}\dfrac{p_1 \dots p_n}{[M := e_1]\,[F := e2]}$ s'interprète de la manière suivante : au-dessus de la ligne se trouve la condition $p_1 \dots p_n$ qui autorise la règle, et en dessous de la ligne se trouvent les informations mises à jour par cette règle pour les différentes variables du système (M,F,. . .). Le nom de la règle est **Rule**.

15. CNF : *Conjunctive Normal Form* (*forme normale conjonctive*)

Les relations sur cette machine sont définies de la manière suivante. La relation $l \prec_{\mathsf{M}} l'$ est vérifiée si le littéral l arrive avant le littéral l' dans M. La relation $level\, l = i$ est vérifiée si et seulement si l arrive dans un niveau de décision i de M. Un modèle propositionnel est noté $\models_p$. L'invariant suivant est conservé durant l'analyse : $F \models_p C$ et $M \models_p \neg C$ quand $C \neq \mathsf{no}$.
Les règles sont décrites dans le tableau 3.2 , et sont appliquées par une machine à états interprétant une formule propositionnelle au format *CNF*, suivant l'ordre de priorité suivant :

1. si n conflits ($n > 0$) ont été trouvés, augmenter n et appliquer **Restart**
2. si une clause est en conflit avec la séquence de littéraux M, appliquer **Conflict**
3. appliquer continuellement **Explain** jusqu'à ce que **Backjump** soit applicable
4. appliquer **Learn**
5. appliquer **Backjump**
6. appliquer **Propagate** pour compléter
7. appliquer **Decide**

La règle **Fail** est appliquée si une assignation de vérité partielle n'est pas dans la séquence de littéraux M. La règle **Forget** est utilisée pour enlever les clauses redondantes, est appliquée après une règle **Backjump**, et est suivie d'une règle **Restart**. Nous proposons maintenant une description simple des règles :

- **Backjump** : Retour en arrière à un certain niveau i de l'analyse
- **Conflict** : Création d'une clause de conflit,
- **Decide** : Transition vers un nouveau littéral,
- **Resolve** : Ensemble de clauses de conflit,
- **Fail** : Détection d'un conflit,
- **Forget** : Pour oublier une clause de conflit,
- **Learn** : Pour retenir une clause de conflit,
- **Propagate** : Compose une clause avec la variable M,
- **Restart** : Remise à zéro des variables M et C.

L'exemple 8 illustre l'utilisation de ces règles.

Exemple 8:
Soit une formule propositionnelle F définie sur un ensemble A, tel que :

$$F := \{A, \{x_1, \overline{x_2} \vee x_3, \overline{x_4}\}\}$$

Backjump	$\dfrac{\mathsf{C} = l_1 \vee \cdots \vee l_n \vee l \quad level\,\bar{l}_1, \ldots, level\,\bar{l}_n \leq i \leq level\,\bar{l}}{[\mathsf{C} := \mathsf{no}] \quad [\mathsf{M} := \mathsf{M}^{[i]} l]}$
Conflict	$\dfrac{\mathsf{C} = \mathsf{no} \quad l_1 \vee \cdots \vee l_n \in F \quad \bar{l}_1, \ldots, \bar{l}_n \in \mathsf{M}}{[\mathsf{C} := l_1 \vee \cdots \vee l_n]}$
Decide	$\dfrac{l \in \mathrm{Lit}(F) \quad l, \bar{l} \notin \mathsf{M}}{[\mathsf{M} := \mathsf{M} \bullet l]}$
Resolve	$\dfrac{\mathsf{C} = l \vee D \quad l_1 \vee \cdots \vee l_n \vee \bar{l} \in F \quad \bar{l}_1, \ldots, \bar{l}_n \prec_{\mathsf{M}} \bar{l}}{[\mathsf{C} := l_1 \vee \cdots \vee l_n \vee D]}$
Fail	$\dfrac{\mathsf{C} \neq \mathsf{no} \quad \bullet \notin \mathsf{M}}{fail}$
Forget	$\dfrac{\mathsf{C} = \mathsf{no} \quad F = G \cup \{\mathsf{C}\} \quad G \models_p \mathsf{C}}{[F := G]}$
Learn	$\dfrac{F \models_p \mathsf{C} \quad \mathsf{C} \notin F}{[F := F \cup \{\mathsf{C}\}]}$
Propagate	$\dfrac{l_1 \vee \cdots \vee l_n \vee l \in \mathsf{F} \quad \bar{l}_1, \ldots, \bar{l}_n \in \mathsf{M} \quad l, \bar{l} \notin \mathsf{M}}{[\mathsf{M} := \mathsf{M} l]}$
Restart	$\dfrac{}{[\mathsf{M} := \mathsf{M}^{[0]}] \quad [\mathsf{C} := \mathsf{no}]}$

TABLE 3.2 – Règles de transitions pour CDCL

La résolution de la formule F par un solveur SAT CDCL est la suivante. L'analyse de la formule commence avec les éléments x_1 et $\overline{x_4}$. La règle applicable à chacun de ces éléments est **Propagate**. Seul l'élément x_1 possède un successeur. On note cette propagation avec un point de décision, et l'analyse arrive au littéral suivant qui est $\overline{x_2}$. La seule règle applicable est **Decide**. x_2 est en disjonction avec le troisième littéral x_3, l'analyse continue à analyser la clause, car la clôture algébrique par rapport à la disjonction de cette clause n'est pas atteinte. Le littéral x_3 est ensuite décrit, il n'existe toujours pas de conflits, la clôture algébrique est atteinte, l'analyse peut appliquer la règle **Propagate**. L'analyseur note cette propagation avec un nouveau point de décision, et arrive au dernier littéral de F. Le littéral $\overline{x_4}$ est défini à l'aide d'une négation, donc l'analyseur applique encore la règle **Decide**. Il n'existe pas d'autre littéral dans F. L'analyse s'arrête à ce moment-là. La formule F est satisfiable (SAT). Nous résumons cette résolution dans le tableau 3.3.

M	F	C	règle
x_1	F	no	**Propagate**
$x_1 \bullet \overline{x_2}$	F	no	**Decide**
$x_1 \bullet \overline{x_2}\ x_3$	F	no	**Propagate**
$x_1 \bullet \overline{x_2}\ x_3 \bullet \overline{x_4}$	F	no	**Decide**

TABLE 3.3 – Résolution de la formule F

3.3.2 Raisonnement automatique à l'aide d'une théorie $\mathcal{T}$

Un solveur *SMT* accepte un ensemble de formules et de requêtes définies dans une théorie $\mathcal{T}$ du premier ordre sans quantificateur, permettant la résolution de ces requêtes au regard de la théorie utilisée et de la base de connaissances construite à l'aide des formules. Le solveur *SMT* construit une abstraction d'un problème, défini à l'aide des formules et des requêtes, par une approximation propositionnelle, et affine de manière incrémentale cette approximation jusqu'à ce qu'elle soit suffisamment précise pour obtenir une solution au problème. La procédure de décision pour une théorie $\mathcal{T}$ utilise deux solveurs : un $\mathcal{T}$-solveur et d'un SAT-solveur. Le $\mathcal{T}$-solveur se concentre sur les informations exprimées dans la théorie, alors que le SAT-solveur s'intéresse aux informations booléennes. Le $\mathcal{T}$-solveur travaille uniquement avec des conjonctions de littéraux. Les réponses attendues de la résolution d'une requête d'un solveur *SMT* peuvent être SAT ou UNSAT. Un solveur *SMT* retourne la réponse « DON'T KNOW »

lorsque la procédure de décision n'a pas pu aboutir. Lorsque la réponse SAT est donnée, le solveur *SMT* peut, lorsqu'il en est capable, retourner un modèle, c'est-à-dire un ensemble d'assignations permettant de satisfaire la formule analysée. Lorsque la réponse UNSAT est retournée, le solveur *SMT* peut, lorsqu'il en est capable, retourner un *sous-ensemble des clauses de conflits qui sont non-satisfiables* (*unsatisfiable core* (*UC*)), plus simplement appelé la *justification*.

Les solveurs *SMT* peuvent être caractérisés en fonction de deux critères : l'approche et le mode d'apprentissage. Deux grandes approches permettent d'utiliser les solveurs *SMT* : l'approche *dynamique* (*Eager*) et l'approche *paresseuse* (*Lazy*) [BDS02]. Nous gardons les termes anglais pour éviter les confusions avec d'autres termes similaires. Nous définissons maintenant quelques termes techniques sur l'utilisation des solveurs *SMT*. Dans l'approche *Eager*, la résolution réalisée par le solveur *SMT* est dirigée par le SAT-solveur. Le $\mathcal{T}$-solveur permet de checker ces prises de décisions, et de générer de nouvelles clauses de conflits si besoin. On dit alors que « la relation du $\mathcal{T}$-solveur avec le SAT-solveur est faible ». Dans l'approche *Lazy*, le $\mathcal{T}$-solveur dirige la résolution, et utilise le SAT-solveur comme une boite noire, c'est-à-dire que le but du $\mathcal{T}$-solveur est de détecter les conflits lors de la résolution d'une requête et d'ajouter des clauses de conflits pour affiner les relations de la solution. On dit alors que « la relation du $\mathcal{T}$-solveur avec le SAT-solveur est fine ». Lorsque le solveur *SMT* essaie de résoudre une requête uniquement en utilisant sa base de connaissances, le solveur *SMT* est en mode d'*apprentissage autonome* (*offline learning*). Lorsque le solveur *SMT* utilise le SAT-solveur pour trouver une solution à un conflit, le solveur *SMT* est en mode d'*apprentissage interactif* (*online learning*). Enfin, les solveurs *SMT* peuvent être utilisés en mode actif, en effectuant une *méthode de va-et-vient* (*back and forth method*). Pour réaliser ce mode actif, il faut qu'un utilisateur soumette une requête à un solveur *SMT*. Si cette requête est UNSAT, alors l'utilisateur peut demander au solveur *SMT* de lui retourner la *justification*. L'utilisateur va alors créer une nouvelle requête en composant la première requête soumise avec la *justification*. L'utilisateur peut recommencer cette procédure jusqu'à ce que la *justification* soit vide, ou que la réponse retournée soit SAT. Nous avons remarqué que les moteurs d'exécution symbolique dynamique utilisent cette procédure lors de l'utilisation d'un solveur *SMT*.

Nous regardons maintenant plus en détail le fonctionnement des solveurs *SMT* où nous réalisons une description simple de chaque approche, puis nous présentons deux optimisations utilisées par les solveurs *SMT*.

3.3.2.1 Approche *Eager*

L'approche *Eager* (figure 3.15) traduit la formule *SMT* d'entrée en une formule booléenne équisatisfiable [16] en une seule étape, à l'aide de plusieurs optimisations, c'est-à-dire que les contraintes ϕ définies dans une théorie complète sont compilées dans une formule booléenne ϕ'. Cette compilation est réalisée en deux étapes : la formule est traduite depuis la théorie *SMT* de la formule d'entrée vers une théorie d'égalité avec fonctions non interprétées [KS16], puis de la théorie d'égalité vers une représentation propositionnelle compréhensible pour un solveur SAT (par exemple par un encodage de *Tseitin*). Des contraintes ϕ_T (appelées contraintes de *Ackermann*) sont aussi ajoutées à la formule ϕ' pour éviter la décohérence. Dans cette approche, le plus grand défi est de trouver une représentation permettant d'obtenir la satisfiabilité des formules le plus rapidement possible (le plus souvent en une seule passe) comme le montre la figure 3.15. C'est pourquoi les solveurs *SMT* exploitant les méthodes *Eager* utilisent des méthodes de traductions robustes et prouvées.

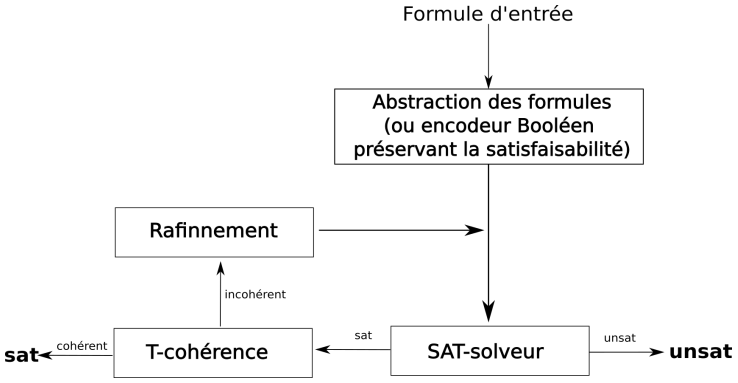


FIGURE 3.15 – Approche *Eager*

Exemple 9:
Une traduction de la formule :

$$\phi \triangleq (x \geq 2) \vee (x \leq 0) \tag{3.2}$$

revient à remplacer $(x \geq 2)$ par un symbole p, et $(x \leq 0)$ par un symbole q. La nouvelle formule devient :

$$\phi' \triangleq (p \vee q) \wedge \phi_T \tag{3.3}$$

16. Soit F une formule au format *CNF* satisfaite par M (ou $M \models F$), alors on dit qu'une formule F' est équisatisfiable à F lorsque F est satisfiable si et seulement si F' est satisfiable.

dans laquelle ϕ_T décrit l'ensemble de la théorie des combinaisons réalisables de $(x \geq 2)$ et $(x \leq 0)$ en termes de p et q. Dans cet exemple, ϕ_T peut prendre la forme de $\neg p \vee \neg q$.

Les solveurs *SMT* basés sur la méthode *Eager* utilisent des solveurs SAT « standards ». Comme cette méthode traduit les formules et requêtes *SMT* depuis la théorie vers une représentation propositionnelle, cette méthode est rapide en pratique. Malheureusement, la quantité de mémoire utilisée par cette méthode peut croître vite, ce qui limite son utilisation.

Exemple de solveur SMT *Eager* : Boolector `Boolector` [NPB14; Nie+18; NPB19] est un solveur *SMT*, utilisant la méthode *Eager*, et supportant la théorie QF-AUFBV, au format *SMT-LIB* [BST+10]. Nous rappelons que la philosophie de la méthode Eager est de travailler dans un espace contraint de solutions avec des limites de solution spécifiques à la théorie. En pratique, `Boolector` traduit les requêtes *SMT* dans son langage interne, nommé BTOR [BBL08]. Ce langage fortement typé [17] est une amélioration au *niveau mots* (*word level*) du langage propositionnel AIGER [18] [BHW11]. Le langage BTOR a pour but de capturer le problème de *vérification de modèles* (*model checking*) [BK08], ce que ne fait pas le format *SMT-LIB*. Le langage AIGER est basé sur l'utilisation de *graphes AND-Inverseur* (*And-Inverter Graphs* (*AIG*)) [19]. Ainsi, lorsque `Boolector` traduit une requête *SMT* dans le langage BTOR, il traduit l'expression en un circuit booléen. De plus, le langage AIGER à l'avantage de permettre la simplification des expressions sans une augmentation du coût mémoire [BB06]. Cette propriété est exploité par `Boolector` pour réduire les expressions avant la traduction au format *CNF*, pour limiter l'utilisation du solveur SAT moderne.

3.3.2.2 Approche *Lazy*

Dans la littérature, le terme *Lazy* pour un solveur *SMT* indique que l'utilisation d'un solveur SAT est dirigée par le $\mathcal{T}$-solveur. La procédure principale est basée sur une extension de la procédure *DPLL* avec une théorie. En pratique, la procédure principale est une extension de la procédure *CDCL* avec une théorie décidable, que nous décrivons un peu plus loin dans cette section. Nous présentons maintenant l'architecture de ce système.
La résolution à l'aide d'un solveur *SMT Lazy* (figure 3.16) est un processus basé sur l'affinement d'une vérification abstraite, respectant les conditions suivantes :

17. *A quantifier-free word-level format for formulas over bit-vectors in combination with one-dimensional arrays*
18. `http://fmv.jku.at/aiger/index.html`
19. Les *graphes AND-Inverseur* sont des *graphes orientés acycliques* (*directed acyclic graphs* (*DAG*))

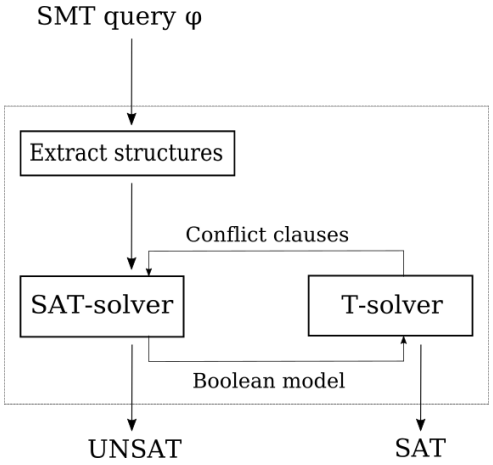


FIGURE 3.16 – Approche *Lazy*

- la formule d'entrée ϕ est abstraite dans une formule propositionnelle ϕ_p ,
- le solveur SAT dénombre les affectations propositionnelles $\alpha_p^1, \ldots, \alpha_p^n$ de ϕ_p ,
- chaque conjonction α^i de $\mathcal{T}$-littéraux est contrôlée pour la $\mathcal{T}$-satisfiabilité,
- dans le cas de la $\mathcal{T}$-non-satisfiabilité de α^i, la formule propositionnelle abstraite doit être affinée.

La figure 3.16 présente un schéma simplifié du fonctionnement de la méthode *Lazy*. Une requête φ définie dans une théorie $\mathcal{T}$ est soumise à un solveur *SMT*. Les relations de la requête sont extraites, puis traduites (après optimisation) en un ensemble de formules propositionnelles. La satisfiabilité de ces formules est ensuite décidée par un *solveur SAT*. Si les formules ne sont pas satisfiables, les clauses de conflits booléennes sont renvoyées vers le $\mathcal{T}$-solveur, qui les traduit dans la théorie $\mathcal{T}$, pour obtenir des $\mathcal{T}$-clauses. Ces $\mathcal{T}$-clauses sont ensuite ajoutées aux autres formules. Ce qui permet de raffiner la requête. Ce processus de raffinement est répété jusqu'à la $\mathcal{T}$-consistance de la représentation booléenne, ou bien lorsqu'il n'existe plus de clauses de conflits.

L'utilisation de l'approche *Lazy* est adaptée aux raisonnements booléens non triviaux (par exemple les grands arbres de recherche), et aux problèmes où la satisfiabilité des littéraux définie dans une théorie est facile. Cette approche n'est pas adaptée pour les problèmes n'ayant pas ou peu de raisonnement booléen (par exemple les petits arbres) ou lorsque la satisfiabilité des littéraux définie dans une théorie est difficile. Pour accélérer l'analyse, différentes procédures de décisions peuvent être combinées, c'est-à-dire que certaines procédures incomplètes (ou sous-procédures rapides) sont appelées avant la procédure principale. Ce qui a pour effet de mettre en place une boucle de raffinement sur les contraintes, et permet au solveur SAT de travailler

$\mathcal{T}-$**Conflict**	$\dfrac{C = \mathsf{no} \quad l_1, \ldots, l_n \in M \quad l_1, \ldots, l_n \models_T \bot}{[C := \bar{l}_1 \vee \cdots \vee \bar{l}_n]}$
$\mathcal{T}-$**Propagate**	$\dfrac{l \in \text{Lit}(F) \quad M \models_T l \quad l, \bar{l} \notin M}{[M := Ml]}$
$\mathcal{T}-$**Explain**	$\dfrac{C = l \vee D \quad \bar{l}_1, \ldots, \bar{l}_n \models_T \bar{l} \quad \bar{l}_1, \ldots, \bar{l}_n \preceq_M \bar{l}}{[C := l_1 \vee \cdots \vee l_n \vee D]}$
$\mathcal{T}-$**Learn**	$\dfrac{\models_\mathcal{T} \exists v(l_1 \vee \cdots \vee l_n) \quad l_1, \ldots, l_n \in L_S \quad v \notin F}{[F := F \cup \{l_1 \vee \cdots \vee l_n\}]}$

TABLE 3.4 – Règles de transitions supplémentaires pour CDCL($\mathcal{T}$)

principalement sur le squelette booléen des contraintes. Les solveurs *SMT* utilisant la méthode *Lazy* possèdent une *interface de programmation d'application* (*application programming interface* (*API*)) [Gan+04] pour réaliser la communication entre le $\mathcal{T}$-solveur et le SAT-solveur, ce qui permet de facilement faire évoluer le solveur *SMT*.

L'algorithme principal de la méthode *Lazy* est basé sur l'algorithme CDCL(T) (algorithme 4). Cet algorithme reprend la même machine à état et règles définies précédemment pour CDCL (table 3.2). Quatre règles sont ajoutées pour aider à la prise de décision dans une théorie ($\mathcal{T}-$**Conflict**, $\mathcal{T}-$**Propagate**, $\mathcal{T}-$**Learn**, $\mathcal{T}-$**Explain**) (tableau 3.4). $\mathcal{T}-$**Conflict** utilise le solveur de théorie pour valider les choix du solveur SAT. $\mathcal{T}-$**Propagate** et $\mathcal{T}-$**Explain** permettent de guider la recherche effectuée par le solveur *SMT*. $\mathcal{T}-$**Learn** permet de retenir les clauses de conflits définis dans une théorie. Les définitions des variables du système sont légèrement modifiées : F contient des clauses définies dans une théorie sans quantificateur, et M est une séquence de littéraux définis dans une théorie sans quantificateur et de points de décisions. L'invariant du système est : $F \models_T C$ et $M \models_p \neg C$ quand $C \neq \mathsf{no}$.

L'exemple 10, initialement proposé par [Tin10], illustre l'approche *Lazy*.

Exemple 10:
Les paramètres d'environnement sont les suivants : la théorie sans quantifieur utilisée est la théorie *égalité avec fonction non interprétée* (*equality with uninterpreted functions* (*EUF*)), le solveur SAT est passif (offline), le solveur *SMT* n'est pas utilisé en mode incrémental. La formule *SMT* ϕ à analyser est la suivante :

$$\phi := g(a) = c \wedge f(g(a)) \neq f(c) \vee g(a) = d \wedge c \neq d \tag{3.4}$$

Algorithme 4 : Algorithme CDCL($\mathcal{T}$) [CS18]

Data : ϕ : SMT query defined over a theory $\mathcal{T}$
Result : SAT or UNSAT

```
1  μ ← [];
2  dl ← 0;
3  while True do
4      res ← theory_and_boolean_propagation(φ,μ);
5      if res = SAT then
6          if all_variables_are_assigned(φ,μ) then
7              return SAT
8          l ← pick_a_branching_litteral(φ,μ);
9          dl ← dl + 1 ;
10         push(μ,l@dl)
11     else
12         (lvl,cls) = theory_and_boolean_conflict_analysis(φ,μ);
13         if lvl < 0 then
14             return UNSAT
15         backtrack(φ,μ,lvl);
16         learn(cls);
17         dl ← lvl
```

Le $\mathcal{T}$-solveur réalise une version abstraite de la formule ϕ, en proposant les atomes suivants :

$$\{x_1\} := \{g(a) = c\}, \{\overline{x_2}\} := \{f(g(a)) \neq f(c)\}, \{x_3\} := \{g(a) = d\}, \{\overline{x_4}\} := \{c \neq d\}$$

La formule propositionnelle F_0 obtenue est définie sur un ensemble A, tel que :

$$F_0 := \{A, \{x_1, \overline{x_2} \vee x_3, \overline{x_4}\}\}$$

Le solveur SAT analyse la formule F_0 et ne trouve pas de conflit. Il retourne SAT. Un modèle booléen est retourné vers le $\mathcal{T}$-solveur $\{x_1, \overline{x_2}, \overline{x_4}\}$ [a]. Le $\mathcal{T}$-solveur compose le modèle au reste de la formule F_0 et trouve un $\mathcal{T}$-conflit. Une nouvelle formule est fabriquée avec la négation du modèle [b] $F_1 := \{A, \{x_1, \overline{x_2} \vee x_3, \overline{x_4}, \overline{x_1} \vee x_2 \vee x_4\}\}$. La formule F_1 est envoyée au solveur SAT. Comme pour la formule F_0 le solveur SAT retourne SAT. Un nouveau modèle booléen est envoyé au $\mathcal{T}$-solveur : $\{x_1, x_2, x_3, \overline{x_4}\}$. Le $\mathcal{T}$-solveur trouve un $\mathcal{T}$-conflit pour ce modèle avec la formule F_0. Le solveur *SMT* fabrique une nouvelle formule avec la négation du dernier modèle : $F_2 := \{A, \{x_1, \overline{x_2} \vee x_3, \overline{x_4}, \overline{x_1} \vee x_2 \vee x_4, \overline{x_1} \vee \vee \overline{x_2} \vee \overline{x_3} \vee x_4\}\}$. La formule F_2 est envoyée au solveur SAT, qui retourne UNSAT car il n'y a pas de nouveaux atomes. Cette formule *SMT* n'est pas satisfiable dans EUF.

a. équivalent au format CNF à $x_1 \wedge x_2 \wedge \overline{x_4}$
b. Les atomes du modèle ne possèdent pas par définition de forme négative.

Exemple de solveur SMT Lazy : Z3 Le solveur *SMT* `Z3` [DB08] utilise la méthode *Lazy*. Nous rappelons que la philosophie de l'approche *Lazy* consiste à étendre le problème SAT vers le problème *SMT*, à l'aide d'une extension de l'algorithme *DPLL*. Pour résoudre les requêtes au format QF-AUFBV, `Z3` utilise l'algorithme CDCL($\mathcal{T}$), avec une règle d'étalement pour les expressions définies dans la Théorie de l'Égalité (*Equality Theory*) et autorise la création de nouveau litteraux dans la théorie [RKG18; 19b], c'est à dire en réalisant de l'*apprentissage interactif* (*online learning*).

3.3.2.3 Optimisations des solveurs SMT

Les solveurs *SMT* possèdent de nombreuses options et optimisations pour être efficaces. Nous en présentons deux : l'utilisation des procédures de décision incomplète, et une amélioration du système de résolution proposé par le solveur *SMT*. Ces optimisations nous ont semblé jouer un rôle dans l'interprétation des prédicats opaques, comme nous le verrons plus tard.

Procédures de décisions incomplètes L'un des avantages de l'utilisation des solveurs *SMT* est de pouvoir utiliser la combinaison de plusieurs procédures de décision. Par exemple, les données manipulées par un programme peuvent être représentées sous forme de *tableaux* (*arrays*). Le raisonnement effectué dans ces tableaux permet d'accélérer en partie la résolution du problème. La combinaison de plusieurs procédures de décisions permet d'obtenir une procédure de décision *incomplète*, mais *efficace*, si elle garantit la propriété de *terminaison*, et pas nécessairement une propriété de *conclusion*. Ces procédures incomplètes sont souvent combinées avec une procédure complète, dans le but de garantir une solution au problème, tout en augmentant son efficacité. Certains solveurs sacrifient la *complétude* (*completness*) et peuvent retourner la réponse « DON'T KNOW », tout en restant capables de résoudre un large panel de problèmes [KS16]. De plus, les procédures incomplètes sont actuellement le seul moyen pour résoudre des problèmes depuis des théories *non décidables* (*undecidables*), par exemple les formules contenant des fonctions trigonométriques.

Système de Résolution étendue L'utilisation de la théorie *égalité avec fonctions non interprétées* à un rôle important dans la résolution de requête par un solveur *SMT Lazy*, et moindre pour une résolution avec un solveur *SMT Eager*. Les travaux de Robere, Kolokolova et Ganesh [RKG18] montrent que l'utilisation d'un solveur SAT incrémental avec un solveur *SMT* permet de créer de nouveaux littéraux définis dans la *théorie égalité avec fonctions non interprétées*. Ce qui permet d'obtenir un système de preuve plus « fort » qu'un système de résolution, c'est-à-dire que ce système de preuve peut effectuer moins d'opérations pour interpréter une requête par rapport un système de résolution. De plus, nous avons cherché à comprendre si toutes les théories composables avec la *théorie égalité avec fonctions non interprétées* autorisaient l'apparition d'un nouveau littéral. Nous n'avons pas trouvé de références ou de preuves claires permettant de répondre à notre interrogation. Nos recherches nous ont amenés à penser que seules les *théories homogènes* (*homogeneous theory*) [20] permettent cette situation. Les chapitres 5 et 6 apportent aussi des éclaircissements à cette question.

Bilan

Les solveurs *SMT* constituent une extension des solveurs SAT, à l'aide d'une théorie complète du premier ordre. Cette théorie peut être combinée avec d'autres théories incomplètes, permettant une meilleure expressivité du problème exprimée dans la requête, et une résolution en pratique

20. La description des théories homogènes est donnée dans le chapitre 5.

plus efficace. Deux solveurs sont présents dans un solveur *SMT* : $\mathcal{T}$-solveur et SAT-solveur. Un solveur *SMT* propose, pour une requête exprimée dans une théorie, une stratégie pour tirer profit de ces deux solveurs. Deux méthodes d'utilisations sont possibles : *Eager* (le SAT-solver dirige la résolution), et *Lazy* (le $\mathcal{T}$-solveur dirige la résolution). Un certain nombre d'optimisations permettent d'améliorer l'efficacité de la résolution de requêtes en fonction de ces méthodes.

Nous venons de voir les mécanismes principaux permettant une analyse symbolique et concrète d'un programme. Nous pouvons maintenant nous reconcentrer sur la présence des prédicats opaques dans un programme, en décrivant les difficultés rencontrées lors de l'analyse d'un prédicat opaque.

3.4 Exemple d'un effet observable lors de l'analyse d'un prédicat opaque

Les sections précédentes nous ont permis de comprendre comment l'analyse d'un programme s'effectue à l'aide d'un moteur d'exécution symbolique dynamique et d'un solveur *SMT*. Un prédicat opaque est analysé maintenant au choix par deux outils `KLEE` ou `angr`. Nous prenons comme exemple de prédicat opaque le produit de deux nombres consécutifs modulo 2, c'est-à-dire que nous regardons la parité du produit de deux nombre consécutifs :

$$x \times (x + 1) = 0 \mod 2$$

Ce prédicat opaque peut être intégré facilement dans un programme. Nous présentons son intégration dans un programme en langage C, avec une variable x typée comme un *entier* (*integer*) de 32-bits. Un exemple de ce programme source est présenté dans le listing 3.6.

```
1 void foo(int x){
2         if(((x*(x+1)%2) != 0 )
3                 assert(false);
4         }
```

Listing 3.6 – Exemple de prédicat opaque dans un programme C

Ce programme est ensuite analysé au choix par `KLEE` ou `angr`. Les deux outils réalisent une analyse symbolique du programme et génèrent une requête dans une théorie décidable : QF-AUFBV ou QF-ABV. Nous présentons rapidement ce que sont les théories QF-AUFBV et QF-ABV. La théorie QF-AUFBV est la composition de la théorie complète QF-BV avec les

théories complémentaires de *tableau* (*array* (*A*)) et de *fonctions non interprétées* (*uninterpreted function* (*UF*)). La théorie QF-ABV est la composition de la théorie QF-BV avec la théorie complémentaire de tableau (*array*). Lors de la résolution d'une requête définie dans la théorie QF-AUFBV, le solveur *SMT* résout d'abord les expressions dans les théories *tableau* et *fonctions non interprétées*, puis traduit les éléments de ces théories vers la théorie QF-BV, c'est-à-dire sous forme de vecteurs de taille fixe (dans le cas de `KLEE` les vecteurs sont des blocs de 8 bits, avec une taille maximum de 32 bits, dans le cas de `angr` ces tailles peuvent être variables). La figure 3.17 illustre ces traductions de théorie.

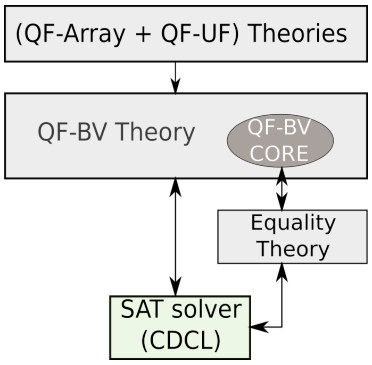


FIGURE 3.17 – Stratégie de l'utilisation de la théorie QF-AUFBV

La résolution d'une requête dans la théorie QF-BV s'effectue classiquement par la méthode d'étalement au niveau bits (*bit-blasting*) des formules. Cette méthode modélise les opérations au niveau bit (fonctions et prédicats) par des circuits booléens. De façon surprenante, la résolution de ce prédicat opaque par un solveur *SMT Lazy* n'est pas satisfiable, ce qui n'est pas le cas avec un solver *SMT Eager*. Si le moteur peut concrétiser le prédicat, alors il trouvera un chemin. Dans le cas contraire, comme le solveur *SMT* ne retourne pas de justification, l'analyse statique ne peut pas aller plus loin.

Nous proposons maintenant une analyse un peu plus approfondie de la résolution de la requête. Autrement dit, c'est une inégalité. Ce prédicat contient une expression algébrique modulaire (à cause de l'opérateur modulo), nous supposons que la représentation de ce prédicat dans la théorie égalité est une expression qui peut être représentée de manière abstraite par un graphe, où chaque nœud est un atome de cette expression, et chaque conjonction entre ces atomes forme les arrêtes. Dans ce cas, l'expression ignore la structure booléenne de l'expression. Ce graphe possède un cycle à cause de l'opérateur modulo, et plus particulièrement un cycle contradictoire à cause de la présence de l'inégalité. Or, d'après les travaux de MEIR et STRICHMAN

[MS05], dans ce contexte, un sous-graphe est non-satisfiable [21] si et seulement s'il contient un cycle contradictoire. Notre prédicat est le plus petit cycle contradictoire pour l'ensemble de définitions $\mathbb{Z}/2\mathbb{Z}$, car x et $x+1$ sont deux générateurs de $\mathbb{Z}/2\mathbb{Z}$ et ne peuvent être décomposé dans cet ensemble. Ainsi, ce prédicat opaque ne peut pas être résolu si la résolution s'effectue dans la théorie de l'égalité.

3.5 Conclusion du chapitre

La déobfuscation automatique à l'aide d'un moteur d'*exécution dynamique symbolique* est un tour de force faisant appel à de nombreux domaines. L'obfuscation d'un programme permet d'entraver le « bon » fonctionnement d'une analyse statique. Pour annuler ces effets, la déobfuscation automatique devient nécessaire. Les méthodes d'analyse statique peuvent être améliorées à l'aide d'un solveur *SMT*, et de différentes stratégies d'explorations. Un ensemble de contraintes permettent de représenter le comportement du programme, et peuvent être résolues à l'aide d'un solveur *SMT*. Ces solveurs permettent de résoudre rapidement des requêtes définies dans une théorie du premier ordre. Nous avons vu un exemple d'application des effets d'un prédicat opaque contenu dans un programme.

21. Un sous-graphe de E est dit satisfiable si et seulement si la formule représentée par ses nœuds est satisfiable.

Deuxième partie

Améliorations de l'analyse des prédicats opaques

Chapitre 4

Un équilibre difficile entre protection logicielle et intégrité de programme

Sommaire

4.1 Introduction 77
4.2 Présentation de la machine 79
4.3 Protection et intégrité du flot de contrôle 81
4.4 Recouvrement du *CFG* 85
4.5 Tests et améliorations 86
4.6 Discussions 96
4.7 Conclusion du chapitre 101

Résumé L'intégrité d'un programme peut être garantie durant son exécution sous certaines conditions, notamment à partir du *graphe de flot de contrôle*. Nous utilisons une simple machine, avec trois *architectures de jeux d'instructions* différents, pour comprendre comment la présence de l'instruction de saut indirect dans un programme permet de garantir cette intégrité ou non. Ce chapitre contient les principaux résultats des travaux réalisés en collaboration avec Ronan LASHERMES (Inria), présentés dans la publication *A Case Against Indirect Jumps for Secure Programs*, à *Software Security, Protection and Reverse Engineering Workshop* (*SSPREW*) 2019 [GL19].

4.1 Introduction

La déobfuscation d'un programme peut s'appuyer sur l'interprétation du *graphe de flot de contrôle* (*CFG*). Dans ce chapitre, nous essayons de comprendre comment la présence de sauts

indirects dans une *architecture de jeu d'instructions* (*instruction set architecture* (*ISA*)) implique la présence de métadonnées pour rendre possible l'extraction d'un *graphe de flot de contrôle*. Pour simplifier la lecture de ce chapitre, nous utilisons parfois le terme de *jeu d'instructions* au lieu d'*architecture de jeu d'instructions*, car nous nous concentrons sur la sémantique de différentes *ISAs*. Nous observons le comportement d'une machine simple qui permet l'utilisation de la propriété d'intégrité d'un programme. Cette propriété est utilisée sur le flot d'un programme et est appelée *intégrité du flot de contrôle* (*control flow integrity* (*CFI*)). Nous nous limitons à l'*intégrité matérielle du flot de contrôle*. Le *graphe de flot de contrôle* d'un programme est précalculé au moment de la compilation du code source en code binaire (ou de la transformation d'un programme). Pour garantir la bonne exécution du programme (c'est-à-dire conformément à ce que le développeur avait prévu), nous utilisons la propriété d'intégrité. Informellement, l'utilisation de l'*intégrité du flot de contrôle* a pour objectif de garantir qu'un attaquant ne peut pas modifier l'ordre des instructions d'un programme. Cela suppose que l'*intégrité du flot de contrôle* capture correctement le comportement du programme, autrement dit qu'elle arrive à extraire l'ensemble des transitions représentées dans le *graphe de flot de contrôle* initialement calculé. Malheureusement, la présence de l'instruction de saut indirect peut empêcher une bonne extraction. De nombreux travaux académiques ont essayé de contourner ce problème, comme le montrent les travaux de Theiling [The00] et ceux de Kinder, Zuleger et Veith [KZV09] pour l'extraction du *graphe de flot de contrôle* en présence de sauts indirects. Mais, ce problème peut aussi être vu comme une solution pour renforcer les méthodes d'*obfuscation logicielle*, en augmentant la difficulté de compréhension des transitions entre les différents *blocs de bases* (*basic blocs*) du *graphe de flot de contrôle* original. En pratique, ce comportement correspond à l'utilisation des *prédicats opaques* pour cacher les conditions de branchement du programme.

Pour montrer l'influence des sauts indirects, nous présentons (section 4.2) une machine simple qui ressemble à ce que l'on peut trouver dans la réalité. Nous définissons l'*intégrité du flot de contrôle* et les *prédicats opaques à trappes* dans la section 4.3. Nous proposons ensuite deux méthodes pour recouvrir le *graphe de flot de contrôle* (section 4.4). Nous y définissons trois *ISAs* (section 4.5), sémantiquement différentes. Ces *ISAs* sont très simples et ont pour seul but la compréhension de l'influence de l'instruction de *saut indirect*. Elles ne sont pas optimisées et c'est pourquoi nous ne les comparons pas à ceux existantes. Nous n'abordons pas le problème de conversion automatique des programmes décrits à l'aide des *sauts indirects* dans nos nouvelles propositions d'*ISAs*. Comme nous le verrons dans ce chapitre (section 4.2 et section 4.6), une telle conversion est impossible en général. Nous réalisons une comparaison théorique à l'aide d'un programme basé sur l'utilisation d'un *distributeur* (*dispatcher*), correspondant au schéma

général d'exécution efficace du *saut indirect*. Nous terminons ce chapitre en discutant les limites de nos propositions (section 4.6). En particulier, nous montrons que la capacité à extraire un *graphe de flot de contrôle* n'est pas transférable dans une *machine virtuelle* (*virtual machine*).

L'ensemble de nos expériences et résultats sont accessibles dans le répertoire *git* à l'adresse suivante : `https://gitlab.com/Artefaritaj/simple_risc`.

4.2 Présentation de la machine

Nous définissons dans cette section la machine simple utilisée, en présentant la sémantique utilisée et en décrivant les différentes *ISAs*. Nous observons comment la sémantique d'une *ISA* influence la propriété d'*intégrité du flot de contrôle*.

4.2.1 Architecture de la machine

Nous utilisons une machine abstraite représentative de ce que nous pouvons trouver actuellement dans les ordinateurs. Les instructions élémentaires exécutables sur notre machine sont inspirées [1] de l'*architecture du jeu d'instructions* (*ISA*) d'un processeur `RISC-V` [2]. Nous allons proposer et étudier trois *architectures de jeux d'instructions* différentes, définies dans la sous-section 4.2.3. Mais avant de présenter plus en détail leurs différences, nous commençons par les éléments communs.

Notre machine est un processeur *RISC* [3] 64-bits : adresses, mots des données et registres sont encodés dans des entiers de 64-bits. La machine possède un état mutable composé de registres et de données (de la mémoire). Les registres peuvent être :

- des registres génériques (de lecture ou d'écriture) $r_1, \cdots, r_{16}$;
- le registre du *pointeur de pile* (*stack pointer* (*SP*)) (de lecture ou d'écriture) pour la pile de données ;
- une constante zéro et des registres $full$ (en lecture seule), où $full$ est définie comme la constante qui respecte la propriété suivante $x \oplus full = \neg x$ ($full$ est la valeur qui contient tous ses bits à la valeur 1) ;

1. Simplifiées aussi dans le but d'obtenir un jeu d'instructions réalisable : nous ne nous occupons pas des problèmes d'encodage des instructions ni des effets de bords.

2. `https://riscv.org/`

3. *RISC* : *Reduced Instruction Set Computer*

- un registre spécial (non lisible et non modifiable directement), appelé pointeur d'instruction (*program counter* (*PC*)), contenant l'adresse mémoire de la prochaine instruction à exécuter.

Comme la mémoire de notre machine est bornée et est adressée par des mots (comme pour des ordinateurs réels), nous pouvons représenter le comportement de notre machine par un *automate linéairement borné* (*linear bounded automaton* (*LBA*)). Ainsi, comme la limite de cette mémoire est explicite, nous ne pouvons pas construire une *machine universelle de Turing* (*universal turing machine*), et tout ce qui est implanté dans cette machine n'est pas *Turing complet* (*Turing completness*) . Cependant, comme chaque processeur réel possède au moins un mécanisme d'arrêt (mais qui n'est pas capturé par nos modèles d'instructions), les programmes implantés dans cette machine correspondent à ceux qui nous sont utiles. Plus précisément, ce sont ceux utilisables dans le monde réel, comme le font remarquer WERNER [Wer97] pour les machines *LBA*.

Tous les registres et toutes les mémoires sont initialisés à la valeur 0 par défaut. Un programme est vu comme un dictionnaire d'instructions : l'adresse est utilisée pour sélectionner une instruction dans le dictionnaire. C'est pourquoi notre architecture est vue comme une *architecture de Harvard* [4] . Dans cette architecture, les instructions et les mémoires sont strictement séparées, ce qui permet à l'unité de contrôle de pouvoir simultanément lire une instruction et lire/écrire des données depuis/vers la mémoire. Ainsi, contrairement à une architecture classique de *VonNeumann*, nous n'autorisons pas qu'une donnée en mémoire puisse modifier une instruction. Le fonctionnement de la machine est décrit dans l'algorithme 5.

Algorithme 5 : Règles pour l'exécution d'un programme

1 Fournir en premier un programme devant être stocké dans une mémoire dédiée (en lecture seule).
2 Régler le *program counter* (*PC*) à l'adresse d'entrée du programme.
3 Puis dans une boucle infinie :
   1. Récupérer la prochaine instruction à l'adresse du *program counter*
   2. Exécuter l'instruction courante en changeant l'état selon la sémantique de l'instruction, ou arrêter la machine pour une instruction `halt`.
   3. Si le *program counter* n'a pas été changé par l'instruction, incrémenter l'adresse.

4. Les architectures de Harvard sont principalement utilisées dans des processeurs de traitement numérique de signal (DSP) ou des microcontrôleurs (PIC ou AVR).

4.2.2 Sémantiques utilisées

Nous présentons une description succincte des instructions que l'on utilisera dans les jeux d'instructions proposés plus loin dans le tableau 4.1. Des pseudo-instructions sont utilisées pour réaliser des opérations de piles en regroupant plusieurs instructions-machine.

4.2.3 Description des *ISAs*

Nous présentons maintenant nos trois *architectures de jeux d'instructions* :

- ISAv1 : autorise l'utilisation de l'instruction de saut indirect.
- ISAv2 : équivalente à ISAv1 sans les sauts indirects.
- ISAv3 : utilise l'ISAv2, avec une instruction de saut indirect restreint : autorisé sous la forme d'une sémantique d'appel/retour avec une pile de retours dédiée, appelée *pile d'appel* (*call stack*).

La présence des *sauts indirects* dans un programme apporte des avantages pratiques comme l'amélioration des temps d'exécutions des programmes, ou le fait de faciliter les compilations de codes sources. Dans toutes les architectures de machines actuelles, ces sauts participent à l'optimisation des exécutions des programmes. La présence de *sauts indirects* ayant des destinations non prédictibles statiquement empêche une extraction précise du *graphe de flot de contrôle*. Ces sauts existent principalement sous deux formes : les *adresses valides* (*forward edges*) (*e.g.* `jump x0`) pour lesquelles le programme saute à l'adresse stockée dans un registre, et les *arêtes de retours* (*backward edges*) (*e.g.* `return`) pour lesquelles le programme revient à la prochaine adresse de la dernière procédure d'appel. Interdire ces sauts peut permettre d'empêcher une extraction du *CFG*. De plus, pour autoriser une plus grande polyvalence d'une *ISA*, gérer les *sauts de retour indirects* (*indirect backward jumps*) d'une manière statiquement prévisible peut être possible. Malheureusement, interdire ces sauts sur les architectures de machines actuelles rend l'exécution des programmes moins rapide (pour des langages réguliers), car certaines actions sont impossibles comme les appels de fonctions. Une solution en pratique pourrait être d'augmenter les contraintes sur le langage.

4.3 Protection et intégrité du flot de contrôle

Nous présentons pus en détail le fonctionnement des *prédicats à trappe* (sous-section 2.3.1), et nous regardons ensuite les solutions pour réaliser l'intégrité du flot de contrôle.

`Integer arithmetic instructions`	$(+, -, *, \div, \mod, <<, >>, \oplus, \wedge, \vee)$ opérations arithmétiques (classiques, modulaires et booléennes) entre deux opérandes d'entiers [5]. Le résultat est retourné dans un registre d'entier.
`Conditional branches`	$(=, \neq, \leq, <, \geq, >)$ opérations logiques (conditionnelles) entre deux opérandes d'entiers. Si la condition est valide, le flot dévie vers une adresse spécifiée, sinon il suit le flot courant.
`Direct jump`	saut direct vers une adresse spécifiée pour la prochaine instruction.
`Call`	saut direct vers une adresse spécifiée et stockage de la valeur courante du *PC* dans un registre [6] ou une pile dédiée [7].
`Return` (seulement pour l'ISAv3)	retourne à l'adresse extraite (*pop*) depuis la pile de retours dédiée.
`Indirect jump` (seulement pour l'ISAv1)	saute à l'adresse indiquée dans le registre.
`Direct load/Direct store`	charge ou stocke la donnée en mémoire vers/depuis un registre.
`Indirect load/Indirect store`	charge ou stocke la donnée en mémoire à partir de l'adresse définie dans un registre.
`Load immediate`	définit la valeur du registre donné (source de constantes).
`Register move`	copie d'un registre dans un autre registre.
`Non deterministic`	charge une valeur non déterministe dans un registre. La sémantique ne permet pas de connaître la source, plusieurs sont possibles : une entrée utilisateur, le résultat d'un véritable générateur de nombres aléatoires, *etc.*
`Halt`	instruction d'arrêt.

TABLE 4.1 – Description des instructions

4.3.1 Protection du flot de contrôle à l'aide de prédicat opaque à trappes

Les *prédicats opaques* (section 2.3) permettent de masquer le flot de contrôle d'un programme. Comme ces derniers ne sont pas *corrects* (*sound*), d'après les travaux de Zobernig, Galbraith et Russello [ZGR17], nous utilisons une autre catégorie de prédicats appelée *prédicats à trappe* (*trapdoor predicate*) comme l'illustre l'exemple donné dans le listing 4.1.

```
1  x     ← user input
2  delta ← p(x)*(h(x+1) xor constant)
3  jump     0x1000 xor delta
```

Listing 4.1 – Masquer le flot de contrôle avec un prédicat à trappe

Dans l'exemple du listing 4.1, la valeur `p(x)` vaut presque tout le temps 0 et le flot du programme saute à la valeur $0\text{x}1000$. Lorsque l'attaquant (ou l'utilisateur) peut rentrer la valeur secrète k dans la variable x, alors le programme saute à l'adresse masquée [8]. Dans ce programme, plusieurs informations ne peuvent pas être trouvées sans la connaissance de k (hormis par une technique de *force brute* (*brute force*)) :

- trouver la valeur de la variable x telle que le saut à une adresse soit différent de la valeur $0\text{x}1000$;
- trouver l'adresse de destination quand le saut vise une adresse différente de $0\text{x}1000$. Nous utilisons la valeur `h(x+1)` puisque la fonction h peut être retrouvée à partir du programme qui calcule la valeur de `p(x)`.

En conclusion, à cause de l'existence des *prédicats à trappe*, l'extraction d'un *CFG* précis peut devenir impossible lorsque l'attaquant n'a pas accès à certaines informations complémentaires. En particulier, la connaissance du programme seul ne suffit pas pour retrouver le *CFG*. Dans l'exemple du listing 4.1, l'extraction du *CFG* doit supposer que l'instruction de saut peut couvrir l'univers des adresses possibles (alors qu'en réalité il ne peut sauter qu'à deux destinations : les adresses définies par les deux conditions).

4.3.2 Intégrité du flot de contrôle (*CFI*)

Un mécanisme d'*intégrité du flot de contrôle* (*CFI*) est chargé d'assurer le « bon » parcours [9] de la structure d'exécution. Pour chaque instruction, il vérifie si la transition à la prochaine

8. Ici : égale à $(0\text{x}1000 \oplus h\,(k+1) \oplus constant)$ où $constant$ est choisie selon la destination désirée
9. Le « bon » parcours correspond à celui initialement prévu par le développeur.

instruction est bien une arête du *CFG* définie par le développeur. De nombreux travaux ont été publiés sur la *CFI*. Nous présentons et détaillons maintenant ceux qui nous semblent les plus pertinents vis-à-vis de notre travail. Burow *et al.* [Bur+17] en proposent une synthèse. La plupart des solutions de *CFI* tentent de vérifier si les sauts peuvent atteindre seulement des *adresses légitimes* (*forward edges*). Un cas spécial peut être observé avec l'instruction `return` qui retourne le flot vers la *routine d'appel* (*backward edges*). Abadi *et al.* [Aba+09] avancent des réalisations de *control flow integrity (CFI)* à l'aide de *fragments de codes* (*code snippets*) pour remplacer les instructions définies dangereuses (*indirect jumps*), c'est-à-dire celles qui ne permettent pas de garantir la *CFI*. Tice *et al.* [Tic+] proposent une solution logicielle qui s'appuie sur le compilateur pour insérer automatiquement les protections appropriées uniquement aux sites des *adresses légitimes* (*forward edges*). En particulier, ils s'attaquent au problème causé par les tables de méthodes virtuelles [10] où un *distributeur* (*dispatcher*) est nécessaire.

La contre-mesure d'usage est de protéger les *arêtes de retours* (*backward edges*) (par exemple les instructions `return`) avec une *pile fantôme* (*shadow stack*) [FS01] pour lesquelles l'appel à la pile est dupliqué, puis de comparer les deux appels. Dans le cas où ils sont différents, une alarme est déclenchée. Cette contre-mesure n'utilise ni de sémantique supplémentaire ni de chemin pour valider l'adresse de retour contre la vraie valeur. Elle correspond à une contre-mesure de *duplication* (*duplication countermeasure*). Ce qui a pour résultat qu'elle ne peut être utilisée que dans quelques scénarios d'attaques spécifiques. Cette contre-mesure ne peut donc pas être considérée comme une solution générale pour garantir l'intégrité du flot de contrôle. Davi *et al.* [DKS14 ; Dav+15] explorent une autre possibilité en ajoutant des instructions à l'*ISA* dans le seul but de valider les appels de fonctions. Dans n'importe quel appel de fonction indirect, le processeur commute dans un état particulier. La prochaine instruction doit être une instruction spéciale `CFIBR label` pour autoriser la poursuite de l'exécution. L'*étiquette* (*label*) utilisée garde la trace des fonctions exécutées.

Pour résister à des modèles d'attaquants plus forts, la *CFI* matérielle devient nécessaire. SOFIA [Rua+16] propose une solution en ce sens, avec un schéma basé sur un *jeu d'instructions aléatoires* (*Instruction Set Randomization* (*ISR*)) : les instructions sont chiffrées avec une valeur qui représente le nœud correspondant dans le *CFG*. Plus précisément, les instructions sont chiffrées au démarrage du programme en utilisant les valeurs précédentes et actuelles du *PC* comme montré dans l'équation (4.1). La nouvelle valeur du *PC* est calculée comme la *disjonction exclusive* (*XOR*) de l'ancienne valeur du *PC* avec le chiffrement de la concaténation de toutes

10. Pour renforcer le polymorphisme d'exécution dans la plupart des langages de programmation (par exemple C++)

les valeurs des pointeurs précédant ce pointeur.

$$i' = E_k(PC_{prev}||PC||...) \oplus i \tag{4.1}$$

Concrètement, une arête du *CFG* doit être valide pour être encodée dans les instructions chiffrées, et son déchiffrement ne doit pas poser de problème, c'est-à-dire que lors de l'exécution l'instruction originale doit être retrouvée. La difficulté survient lorsque deux prédécesseurs [11] existent pour deux instructions. Une *astuce* (*hack*) alors est proposée pour résoudre cette difficulté.

Enfin, Hiscock, Savry et Goubin [HSG17] explorent comment réaliser proprement un schéma de *jeu d'instructions aléatoires* (*instruction set randomization* (*ISR*)) pour assurer la *CFI* en discutant et proposant plusieurs stratégies de chiffrement.

Pour toutes ces propositions, le système doit être capable de contrôler le flot du programme pour qu'il suive un *CFG* prédéterminé. Puisque les prédicats à trappe ne le permettent pas en général, dans la prochaine sous-section nous proposons de modifier la sémantique de l'*ISA* pour rendre l'extraction du *CFG* facile et précise pour tous les programmes.

4.4 Recouvrement du *CFG*

Le problème du recouvrement d'un *CFG* consiste à retrouver le graphe de flot de contrôle d'un programme à partir d'une représentation de ce même programme. Dans cette section, nous décrivons notre stratégie de recouvrement d'un *CFG*, qui s'effectue à l'aide de deux algorithmes différents :

- *Structurel* : les successeurs d'une instruction sont naturellement tous ceux qui peuvent être atteints par les instructions sémantiques. Un *CFG* est déduit à partir de la structure du programme, et non des données qu'il contient. En particulier, tous les successeurs d'un *saut indirect* (*indirect jump*) sont des instructions du programme. L'algorithme d'extraction est décrit dans l'algorithme 6. Cet algorithme accepte une adresse d'entrée du programme, ainsi qu'un dictionnaire d'instructions, et retourne un graphe équivalent au *CFG*. L'adresse d'entrée est stockée dans une pile. Puis, pour chaque adresse d'une instruction contenue dans cette pile, l'algorithme calcule tous les successeurs accessibles les un après les autres, et les stocke dans la pile. L'algorithme recommence tant que la pile n'est pas vide.

11. deux arêtes

- *Trace* : extraire précisément le *CFG* avec la sémantique d'*appel* (*call*) ou de *pile* (*stack*) de l'ISAv3 est possible. En plus des informations structurelles, nous traçons la pile de retours et calculons le taux de recouvrement [12] des nœuds de tous les états de la pile. En d'autres termes, si la même procédure est appelée depuis deux adresses différentes, l'instruction `return` indique au flot deux adresses différentes. Elles peuvent être trouvées en déterminant toutes les valeurs de la pile de retours dans l'analyse de notre *CFG*. L'algorithme principal est décrit dans l'algorithme 7, qui fait appel à l'algorithme 8 pour calculer la fonction de hachage.

Avoir un meilleur algorithme d'extraction de *CFG* en la présence de sauts indirects est faisable, comme le propose la solution d'analyse statique de KINDER, ZULEGER et VEITH [KZV09]. Malheureusement cette solution n'est pas générale : les *fragments de code* (*snippet*) présentés dans la sous-section 4.3.1 ne sont pas gérés, car cette solution ne prend pas en compte de codes automodifiants. Ainsi, ces fragments ne peuvent être intégrés dans le contexte d'une exécution sécurisée.

Nous proposons ici un algorithme d'extraction de *CFG* plus simple que celui proposé par KINDER *ET AL.* : seule une analyse du flot de contrôle est réalisée. L'analyse du flot de données peut améliorer légèrement la précision du *CFG* extrait, si plusieurs conditions de branchement sont toujours vraies ou toujours fausses pour toutes les exécutions [13]. Mais un tel cas correspond d'une certaine manière à une défaillance du compilateur : cette invariance aurait dû être détectée. Cela aurait rendu l'exécution et l'interprétation du programme plus simples.

4.5 Tests et améliorations

Dans la sous-section 4.2.2, nous avons défini nos trois *ISAs*. Nous pouvons maintenant vérifier si nos propositions permettent effectivement une extraction statique facile et précise du *CFG*. Dans le même temps, nous mesurons les performances des différences des deux tests :

- un chiffrement à l'aide de l'*AES* est un programme simple. L'exécution de ce chiffrement nécessite des opérations d'accès mémoire et beaucoup d'opérations arithmétiques (tableau 4.2).
- les *distributeurs* (*dispatchers*) sont les principaux arguments pour justifier l'utilisation des sauts indirects : nous analyserons leurs impacts (négatifs) dans les *ISAs* v2 et v3,

12. Le taux de recouvrement correspond au ratio entre les instructions qui sont trouvées par l'algorithme d'extraction et les vraies instructions du programme.

13. Autrement dit, ces conditions peuvent être vues comme des tautologies.

Algorithme 6 : Algorithme d'extraction du CFG structurel

```
   Data : E : Address (entry address), P : Dictionary<Address, Instruction> (program)
   Result : CFG : Graph<Address> (control flow graph)
   /* list of addresses to analyze                                          */
1  addresses_buffer ← [E];
   /* addresses already analyzed                                            */
2  analyzed ← {};
3  while addresses_buffer is not empty do
       /* we now analyze instruction at new address                         */
4      current_address ← pop from addresses_buffer;
5      push current_address to analyzed;
       /* find successors, the set of addresses that can follow this address. Here
          the successors are chosen according to the nature of the instruction only
          (branches have 2 successors, etc.).                               */
6      successors ← successor_analysis(program P, current_address);
7      for successor ∈ successors do
           /* The add edge method adds nodes too if necessary               */
8          add edge to CFG : current_address → successor;
9      successors_to_analyze ← filter successors to keep only addresses not in analyzed;
10     append successors_to_analyze to addresses_buffer;
11 return CFG;
```

Algorithme 7 : Suivi de l'algorithme d'extraction du CFG, utilise l' algorithme 8 pour la fonction de hachage `ReturnStack`

Data : E : Address (entry address), P : Dictionary<Address, Instruction> (program)
Result : CFG : Graph<Address> (control flow graph)

```
/* list of pairs (addresse to analyze, corresponding ReturnStack)            */
addresses_buffer ← [(E, empty ReturnStack)];
/* the data contained in the pair (Address, ReturnStack) is hashed for easier
   handling and for dealing with recursion                                   */
/* hashes already analyzed                                                   */
analyzed ← {};
while addresses_buffer is not empty do
    /* we now analyze instruction at new address                             */
    (current_address, current_stack) ← pop from addresses_buffer;
    push (hash(current_address) ⊕ hash(current_stack)) to analyzed;
    /* find successors, the set of pairs (addresses, corresponding return stack)
       that can follow this (address, return stack) pair. Here the successors
       are chosen according to the nature of the instruction and the values in
       the return stack.                                                     */
    successors ← successor_analysis(program P, current_address, current_stack);
    for (succ_address, succ_stack) ∈ successors do
        /* The add edge method adds nodes too if necessary                   */
        add edge to CFG : current_address → succ_address;
    successors_to_analyze ← filter successors to keep only pairs (address, stack) such
     that (hash(add) ⊕ hash(stack)) is not in analyzed;
    append successors_to_analyze to addresses_buffer;
return CFG;
```

Algorithme 8 : Algorithme de hachage `ReturnStack`

Data : RS : Vec<Address> (return stack)
Result : H : integer (hash value)

```
/* The objective is to transparently deal with recursion. If a repeated pattern
   is found at the end, one repetiton is not taken into account. hash(A) =
   hash(AA), hash(AB) = hash(ABAB), hash(ABC) = hash(ABCABC), ...            */
max ← length(RS);
if max > 1 then
    stack_size ← length(RS);
    pattern_size ← 1;
    while pattern_size * 2 <= stack_size do
        /* detect pattern repetition of size pattern_size                    */
        if RS[(stack_size − pattern_size) .. stack_size] ==
         RS[(stack_size − 2 ∗ pattern_size) .. (stack_size − pattern_size)] then
            /* recursion found                                               */
            max ← max - pattern_size;
            break;
        else
            pattern_size ← pattern_size + 1;
H ← 0;
for i ← 0 to max − 1 do
    H ← hash(H | RS[i]);
return H;
```

c'est-à-dire une analyse théorique d'un schéma du distributeur.

4.5.1 Exemple avec l'*AES*

Pour nos tests, nous avons développé à la main dans nos trois langages assembleurs l'algorithme de chiffrement symétrique *AES* [14], en respectant les caractéristiques du document FIPS-197 du NIST [15]. Puis nous avons effectué les tests pour 10 000 exécutions. La quantité d'utilisation des instructions des différentes *ISA* (ISAv1, ISAv2 et ISAv3) est illustrée par les données du Tableau 4.2.

	ISAv1		ISAv2		ISAv3	
Type	Count	Ratio	Count	Ratio	Count	Ratio
Arithmetic	10 011	0,44	9 978	0,41	9 825	0,44
LoadImmediate	5 232	0,23	6 767	0,27	5 218	0.23
IndirectLoad	2 650	0,12	2 628	0,11	2 554	0.11
IndirectStore	2 464	0,11	2 441	0,10	2 368	0.11
DirectJump	668	0,03	719	0,03	668	0,03
IndirectJump [16]	616	0,03	X	X	616	0,03
ConditionalBranch	601	0,03	1 703	0,07	601	0.03
RegisterMove	370	0,02	370	0,02	370	0,02
NonDeterministic	2	0,00	2	0,00	2	0,00
Halt	1	0,00	1	0,00	1	0,00
Total	22 615	1,00	24 609	1,00	22 223	1,00

TABLE 4.2 – Ensemble d'instructions pour un AES pour chaque ISA.

Ces langages assembleur sont aussi utilisés dans les listings 4.2, 4.3, 4.4, 4.5 et 4.7. Dans ces langages assembleurs, la sémantique utilisée capture quasiment tous les comportements des instructions-machine. Le symbole `@` est utilisé pour indiquer la valeur utilisée à l'adresse présente dans le registre. Le symbole `#` est utilisé pour indiquer les valeurs littérales. Enfin, le symbole `!` est utilisé pour indiquer les symboles globaux (par défaut, les symboles sont définis seulement dans un fichier).

```
1 //x1 byte value to modify
2 !xtimes:
3     // save context
4     push    x2, x4
```

14. *AES* : *Advanced Encryption Standard*

15. https ://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf

```
5
6       // x1=x1*2
7       load    x4, #0x1
8       sla     x1, x1, x4
9
10      // test modulo
11      load    x4, #0x100
12      and     x2, x1, x4
13      beq     end, x2, zero
14
15  sub_modulo:
16      load    x4, #0x11B
17      xor     x1, x1, x4
18
19  end:
20      // restore context
21      pop     x4, x2
22      //return
23      jump    x14
```

Listing 4.2 – Implémentation assembleur de Xtimes (sous fonction de l'AES) avec l'ISAv1

4.5.2 Résultats et observations

Chacune des trois *ISA* est analysée en fonction de son efficacité temporelle d'exécution et de la surface de *CFG* extraite sans annotations, comme indiqué dans le tableau 4.3. Pour l'ISAv2, toutes les instructions `return` ont été remplacées par des *distributeurs* (*dispatchers*) (*cf.* sous-section 4.5.3). Une autre possibilité aurait été d'annoter toutes les instructions, mais cette solution ne peut pas être envisageable à l'échelle de tous les programmes.
Dans le tableau 4.3, nous donnons les indicateurs de notre programme. Nous définissons l'efficacité pour la moyenne et l'écart-type des $10\ 000$ exécutions. Le taux de recouvrement des nœuds du *CFG* est défini par le rapport entre le nombre de nœuds dans le *CFG* qui ont été touchés durant une exécution par rapport à celle réellement définie dans le *CFG*. Nous définissons de la même manière le taux de recouvrement des arêtes. Par exemple : un taux de recouvrement des nœuds de $6{,}4\%$ veut dire que seulement $6{,}4\%$ des nœuds ont été effectivement suivis durant

une exécution, et que le *CFG* obtenu est imprécis. Autrement dit, cela indique qu'un nombre important de transitions définies dans le *CFG* capturé sont considérées comme valides alors qu'en réalité elles ne recouvrent pas le *CFG* recherché, et ne sont donc pas valides.

	CFG extraction	Mean duration	Std deviation	CFG nodes count	nodes coverage	CFG edges	edges coverage
ISAv1	structural	22 635,0	24,1	778	99,6%	12 437	6,4%
ISAv2	structural	24 605,2	24,1	816	99,1%	836	99,0%
ISAv3	structural	22 230,7	23,8	747	99,7%	11 942	6,4%
ISAv3	tracking	22 230,8	24,0	745	100%	764	100%

TABLE 4.3 – Évaluation des performances des ISAs et du recouvrement du CFG pour une implémentation de l'AES

La lenteur de l'implémentation avec ISAv2 est due à l'utilisation de l'instruction de retour `return`. Cette dernière est remplacée par un distributeur revenant à la position correcte. L'ISAv1 est un peu plus lente que l'ISAv3 puisque la manipulation des piles (`push` et `pop`) est explicite dans l'ISAv1 au lieu d'être implicite comme dans l'ISAv3.

Concernant l'extraction du *CFG*, la structure d'extraction des ISAv1 et ISAv3 est imprécise. Pour l'ISAv3-tracking, le problème vient des instructions de retour `return`. L'algorithme ne peut pas déduire de conséquence logique dans ce saut sans un apport d'informations extérieures, et il suppose que toutes les instructions ont un potentiel successeur. En pratique, ces deux cas reviennent à vouloir appliquer un *algorithme glouton* (*greedy algorithm*) pour réaliser l'extraction du *CFG*. Comme ce type de problème est difficilement résoluble en général, l'extraction est difficilement praticable.

Mais pour l'ISAv2 avec une extraction structurelle et l'ISAv3 avec l'extraction de traces, le recouvrement du *CFG* est extrêmement précis (respectivement 99,1% et 100%). À cause de la sémantique de l'*ISA*, le recouvrement de l'ISAv2 n'est pas parfait : les instructions inaccessibles d'arrêt `halt` pourraient être ajoutées pour gérer les cas d'erreurs.

Ce test nous montre que pour les propositions des *ISAs* v2 et v3, le *CFG* obtenu est extractible avec seulement une perte de 9% des performances pour l'ISAv2 et un gain de 2% des performances pour l'ISAv3. Ainsi, nos propositions d'*ISAs* permettent bien de réaliser une extraction facile et précise du *CFG*, pour obtenir un *CFG* équivalent à l'original.

4.5.3 Recouvrement des distributeurs

4.5.3.1 Distributeur

Un *distributeur* (*dispatcher*) est une instruction contenant une structure logique conditionnelle particulière qui permet d'accéder à d'autres instructions en fonction de la valeur de la donnée soumise au distributeur. Cette instruction est utilisée pour réaliser les fonctions d'appels dans une bibliothèque, comme un résultat d'un système d'appel ou comme une façon de mettre en œuvre une liaison dynamique dans des langages-objets. Cette configuration est particulièrement dangereuse par rapport aux informations de sécurité comme nous pouvons le voir dans la section 4.6, mais l'utilisation de sauts indirects permet d'avoir une exécution efficace. Par conséquent nous nous attendons à ce que nos propositions d'*architecture de jeu d'instructions* modifient le comportement du distributeur.

```
1   //x1 contains the address to branch to
2   //(may be the result of pointer arithmetic)
3   call x1
4 dispatcher_end:
5   //continue
6
7   //...
8 procX:
9   push x14
10  //...
11  pop x14
12  jump x14
```

Listing 4.3 – Motif du répartiteur ISAv1

```
1   //x1 contains the value that decides where to branch
2   load x15, #0
3   beq proc1, x1, x15
4   jump proc2
5
6 dispatcher_end:
7   //continue
8
```

```
  //...
procX:
  //...
  jump dispatcher_end
```

Listing 4.4 – Motif du répartiteur ISAv2

```
  //x1 contains the value that decides where to branch
  load x15, #0
  beq call_proc1, x1, x15
  jump call_proc2
call_proc2:
  call proc2
  jump dispatcher_end
call_proc1:
  call proc1
  jump dispatcher_end
  //...
dispatcher_end:
  //continue

  //...
procX:
  //...
  return
```

Listing 4.5 – Motif du répartiteur ISAv3

Pour chaque architecture proposée, nous évaluons le nombre d'instructions nécessaires pour représenter le *distributeur* (*dispatcher*) comme une fonction du nombre de procédures p devant être disponibles. La synthèse de l'ensemble de ces calculs se trouve dans le tableau 4.4.

	Branch logic (red)	Call logic (green)	Return logic (blue)	Total
ISAv1	0	1	$3 \cdot p$	$3 \cdot p + 1$
ISAv2	$2 \cdot p$	0	p	$3 \cdot p$
ISAv3	$2 \cdot p$	$2 \cdot p$	p	$5 \cdot p$

TABLE 4.4 – Nombre d'instructions par répartiteur (pour $p > 3$)

	Branch latency (red)	Call latency (green)	Return latency (blue)	Total
ISAv1	0	1	3	4
ISAv2	$2 \cdot \lceil \log_2(p) \rceil + 1$	0	1	$2 + 2 \cdot \lceil \log_2(p) \rceil$
ISAv3	$2 \cdot \lceil \log_2(p) \rceil + 1$	2	1	$4 + 2 \cdot \lceil \log_2(p) \rceil$

TABLE 4.5 – Latence de répartiteur

La *latence de branche*, mesure du temps pris pour traiter l'appel particulier de procédure, est donnée dans le tableau 4.5. Les valeurs des ISAv2 et ISAv3 peuvent être prouvées par récurrence. En effet, pour un p donné, le branchement peut être réalisé en utilisant une branche (c.-à-d. deux instructions) et deux schémas pour les valeurs $\lceil \frac{p}{2} \rceil$ et $\lfloor \frac{p}{2} \rfloor$.
Ce qui nous donne $Latency\,(p) = 2 + Latency\left(\lceil \frac{p}{2} \rceil\right)$.

Si nous admettons que $\forall l < p$, la relation $Latency\,(l) = 2 \cdot \lceil \log_2(l) \rceil + 1$ est vérifiée (manuellement vérifiée pour les premiers l), alors :

$$\begin{aligned} Latency\,(p) &= 2 + 1 + 2 \cdot \lceil \log_2(\lceil \frac{p}{2} \rceil) \rceil, \\ Latency\,(p) &= 1 + 2 \cdot (1 + \lceil \log_2(p) - 1 \rceil)\,, \\ Latency\,(p) &= 1 + 2 \cdot \lceil \log_2(p) \rceil \quad . \end{aligned}$$

Dans le cas où toutes les procédures appelées par le retour du distributeur proviennent du même emplacement, nous pouvons observer que l'ISAv2 est légèrement meilleure que l'ISAv3 pour les deux métriques (coût de l'instruction et coût de la latence). Ce qui ne correspond pas à la vraie raison de la prééminence des sauts indirects : à cause de la répartition de la faible latence, elle est **indépendante du nombre de procédures à appeler**. Plus généralement, le nombre important de branchements des *ISAs* v2 et v3 n'est pas compatible avec une exécution spéculative efficace : la suppression des sauts indirects pénalise les distributeurs. Les sauts indirects suivent une structure de contrôle de branches de sorties (*fanout*) élevé : une instruction peut avoir un ensemble de successeurs possible. Par exemple, dans le cas des *ISAs* v2 et v3, une instruction possède deux successeurs au moins.

4.5.3.2 Amélioration du recouvrement

Afin de trouver un meilleur compromis entre performances et recouvrement/découverte du *CFG*, une nouvelle instruction peut être ajoutée : la branche conditionnelle qui possède n sorties (*the n-fanout conditional branch*).

```
1  nbranch x1, #3
2  jump    proc1
3  jump    proc2
4  jump    proc3
5  jump    error_handling
```

Listing 4.6 – Illustration d'un branchement conditionnel à n-sorties

Le registre choisi doit contenir la valeur entière i dans l'intervalle $[\![0, n-1]\!]$ (n est un littéral constant présent dans l'instruction). Un saut implicite est réalisé à l'adresse : (`AddressOf(branch)` $+1+i$). Si le registre contient une valeur illégale, nous sautons à l'adresse de gestion d'erreur (*error handling*) à (`AddressOf(branch)` $+1+n$). Un exemple d'usage peut être trouvé dans le listing 4.6.

Avec cette nouvelle instruction, la latence possède un taux de croissance plus faible. Elle est de l'ordre de $\mathcal{O}\left(\log_n(p)\right)$, c'est-à-dire que le taux de croissance de la latence tend à devenir plus faible, voire à devenir nul lorsque la valeur de n est égal à p.

4.6 Discussions

4.6.1 Limites de la *CFI*

La capacité à extraire précisément chaque *CFG* limite la portée de la virtualisation comme mécanisme de défense : le mécanisme interne de la machine virtuelle ne peut être caché. Mais comme démontré dans cette section, il est encore possible de cacher la structure de contrôle dans le domaine des données.

4.6.1.1 Machines virtuelles

Garantir la *CFI* est particulièrement important pour réduire les attaques *ROP* [Roe+12] dans lesquelles un attaquant modifie la structure de contrôle depuis un programme valide pour obtenir un comportement malveillant. Alors la *CFI* ne suffit pas. En effet, l'existence de l'émergence de machines virtuelles, en particulier les machines bizarres (*weird machines*) [Dul19], limite les garanties offertes par la *CFI*. Nous illustrons dans le listing 4.7 le fonctionnement d'une machine virtuelle *Subleq*, compatible avec toutes nos *ISA*.

```
1 // data memory is filled with user defined values
2
```

```
// initialization
// virtual program counter
// VPC(x13) = 0
  load  x13, #0

//exec one subleq instruction
subleq:
//read operands from data memory
  move  x1, x13
  load  x15, #1
  add   x13, x13, x15
  move  x2, x13
  add   x13, x13, x15
  move  x3, x13
//increment for next instruction
//if no jump
  add   x13, x13, x15

//subleq execution
  load  x4, @x1
  load  x5, @x2
  sub   x6, x5, x4
  store @x2, x6
  ble   ijump, x6, x0

//start next instruction
  jump  subleq

//virtual indirect jump
ijump:
  move  x13, x3
//start next instruction
  jump  subleq
```

Listing 4.7 – Machine virtuelle subleq

Les machines *Subleq* (*Subtract and branch if Less than or EQual to zero*) ont été prouvées équivalentes à des ordinateurs universels [MP88]. Notre *machine virtuelle* (*VM*) *Subleq* aurait un simple flot de contrôle avec seulement des sauts indirects (définissable avec nos *ISAs*). Les instructions des *machines virtuelles* sont définies par les mémoires de données (*data memory*) et les sauts indirects dans les domaines des données (*data domain*). Ce programme est un exemple dans lequel même si la *CFI* est garantie, nous ne pouvons pas garantir l'ensemble de l'exécution. En d'autres termes, les propriétés de sécurité telles que la *CFI* ne sont pas automatiquement transposées dans une *machine virtuelle* : si un attaquant remplace un mot dans les mémoires de données, il peut *détourner* (*hijack*) le flot de contrôle.

Afin de respecter notre schéma proposé d'une *ISA* qui contient l'information sémantique à propos du flot de contrôle, nous devons concaténer des données supplémentaires aux pointeurs utilisés pour les accès mémoires directs. Il existe une proposition présentée dans CHERI [Woo+14]. Malheureusement, ces nouvelles instructions peuvent ralentir l'émergence des *machines virtuelles*, mais ne permettent pas d'exécuter correctement la *CFI*. Les instructions ci-dessous peuvent remplacer des accès mémoires indirects pour restreindre leurs portées :

- `BoundedIndirectLoad Rd, Rs, B1, B2` : charge la valeur présente en mémoire à l'adresse contenue `Rs` dans le registre `Rd` si l'adresse est comprise entre les bornes `B1` et `B2` [17].
- `BoundedIndirectWrite Rd, Rs, B1, B2` : même principe, mais stocke à l'adresse `Rd` la valeur de registre `Rs`.

4.6.1.2 Orientation de notre modèle de simplifications

Nous regardons ici les limites du modèle simplifié de machine par rapport à des machines réelles.

Les machines présentées dans la section 4.2 sont des versions simplifiées des machines actuelles. En particulier, les machines réelles implémentent des mécanismes dans lesquels le flot de contrôle n'est pas défini par les instructions : *interruption* (*interrupt*) et *trappe* (*trap*). Une *interruption* est un changement du flot de contrôle à cause d'un événement externe : par exemple quand un nouvel octet est arrivé dans un bus série, une interruption est levée pour dévier le flot de contrôle à la sous-routine dédiée, *quelque soit l'instruction précédente*. Une *trappe* est une

17. définissant des valeurs littérales

interruption spéciale réalisée par une instruction [18]. Les *trappes* sont notamment utilisées pour transférer le contrôle depuis l'*espace utilisateur* (*user land*) vers le *noyau* (*kernel*).

Ces mécanismes ne sont pas capturés par nos modèles ni par la plupart des mécanismes de *CFI* présentés dans la section 4.3. Modéliser la sécurité de ces transferts de contrôle est un frein majeur qui n'a pas été encore levé. Par exemple, comment peut-on garantir la *CFI* en présence de la *page de défaut dans une machine bizarre* (*page-fault weird machine*) [Ban+13] ?

Une autre limite vient du fait que nous utilisons une architecture de *Harvard*. Bien que la distinction entre les architectures de *Harvard* et de *Von Neumann* est faible en théorie, nous ne pouvons pas ici modifier les instructions de notre machine après que le programme initial a été écrit. De plus, dans ce cas, l'installation de nouveaux programmes après la programmation initiale n'est pas possible. Ce qui limite fortement un ordinateur universel . La raison étant que n'importe quelle modification d'instruction peut potentiellement casser la *CFI*, et cela est plus difficile à modéliser.

4.6.2 Programmes acceptés et rejetés

Afin d'exprimer un programme avec les *ISAs* v2 ou v3, l'extraction du *CFG* doit être effectuée au moment de la compilation (ou de la transformation de code). Dans nos nouvelles *ISAs*, le nouveau programme reflétera la qualité du *CFG* précalculé. Tout ce qui suit l'extraction du *CFG* devient alors trivial (*cf.* les algorithmes 6 et 7) car équivalent au *CFG* précalculé. De plus, en pratique, nous observons que si l'extraction du *CFG* n'est pas précise, le programme obtenu devient extrêmement inefficace (à la fois en taille et en temps d'exécution) à cause de la présence de grands *distributeurs* (*dispatchers*). Pour tous ces éléments, nous pouvons dire qu'un programme sans saut indirect permet d'obtenir une équivalence d'exécution de la structure de son *graphe de flot de contrôle* précalculé. De plus, la capacité à exprimer un programme dans de nouvelles *ISAs* est aussi équivalente à la capacité d'extraction d'un *CFG* complet au moment de la compilation (les arêtes valides sont toujours prises à l'exécution (*runtime*)).

4.6.2.1 Programmes acceptés

Comme discuté dans la sous-section 4.3.1, il existe des programmes qui ne peuvent pas être exprimés efficacement dans nos *ISAs* v2 ou v3. Ici, le problème est de savoir comment écrire l'extrait (*snippet*) montré dans le listing 7.2 dans une *ISA* sans *sauts indirects*. Si le programme

18. L'instruction `trap` dévie le flot de contrôle vers un emplacement défini par construction (*implementation-defined*)

avait une taille de n instructions, nous pourrions écrire la nouvelle version en calculant en premier les adresses de destinations comme dans la version initiale, puis les ajouter à cette adresse. Comme vu dans le tableau 4.4, le nouveau programme aura au moins des instructions de taille $3 \cdot n$ avec l'ISAv2 et l'ISAv3 (n pour la version initiale, $2 \cdot n$ pour pouvoir accéder à toutes les instructions initiales). Nous ne connaissons toujours pas la valeur secrète qui provoque une destination de branchement différent ou qu'elle est la nouvelle destination. Mais dans cette nouvelle version, la structure du programme rend explicites toutes les branches possibles.

Un programme peut être converti dans l'un de nos *ISAs* si et seulement si sa taille est finie et la disposition de la mémoire est connue (nous considérons seulement les sauts à des adresses d'instructions valides). De manière équivalente pour un nouveau programme, ce dernier peut être exprimé dans une nouvelle *ISAs* si toutes ses instructions sont connues au moment de la compilation. Malheureusement, la conversion d'un programme dans nos *ISAs* ne permet pas de conserver le comportement original du programme (en temps et en mémoire), ce qui rend toutes conversions inefficaces.

En particulier, les tables de méthodes virtuelles peuvent être écrites dans les ISAv2 et ISAv3. Ces tables sont utilisées dans les langages-objets pour traiter la liaison dynamique : en fonction du type des appels, une fonction différente (à une adresse dédiée) est appelée. Les tables de méthodes virtuelles peuvent être remplacées par les distributeurs puisque toutes les destinations sont connues au moment de la compilation.

4.6.2.2 Programmes rejetés

Plusieurs modèles de programmation permettent d'empêcher l'extraction d'un *CFG* complet au moment de la compilation. Par exemple une application capable de charger dynamiquement des plug-ins pour des fonctionnalités supplémentaires. Quand l'application est compilée, les instructions du plugin ne sont pas connues. Si nous interdisons les sauts indirects, l'application ne peut plus être interrompue par une instruction d'interruption lors de son exécution dans la machine. De la même manière, le noyau ne peut plus lancer une nouvelle application non connue au moment de la compilation. Nous devons recompiler notre noyau pour chaque nouvelle application ajoutée. En fait, la question est comment exécuter un programme après compilation, puisque le nouveau programme n'est pas connu avant, et que le point d'entrée des programmes n'est pas fixé à l'avance. Du point de vue de la sécurité, certains schémas sont dangereux. Par exemple, l'exécution d'un *plugin* est équivalente à l'exécution d'instructions inconnues au moment de la compilation de l'application depuis le contexte de l'application.

4.6.2.3 Les sauts indirects sont nécessaires

Démarrer une nouvelle application inconnue depuis le *noyau* (*kernel*) semble plus acceptable que de lancer un *plugin* depuis une application. Cela est dû au fait que dans le premier cas l'application possède moins de privilèges que le *noyau*. Ces exemples nous donnent des pistes de solutions à notre problème. Les *sauts indirects* sont absolument requis pour avoir des fonctionnalités non triviales. Mais un *saut indirect* est aussi équivalent au basculement dans un nouvel environnement sécurisé. Par conséquent, dans un jeu d'instructions sécurisé, si nous souhaitons enlever les *sauts indirects* comme dans notre nouvelle *ISA*, nous devrions aussi introduire des domaines de sécurité matérielle. Ce que nous pouvons définir comme des instructions et ce qui peut être défini comme des données est lié à un domaine de sécurité. Ainsi, nous pouvons dire qu'il existe dans ce domaine une limite permettant de transformer des données de la mémoire en instructions. Avec cette nouvelle fonctionnalité, un saut indirect peut être autorisé lors du passage à un nouveau contexte. Le nouveau saut ne serait pas lié à une instruction particulière, mais au nouveau point d'entrée du domaine de sécurité, capable de vérifier la validité du nouveau flot de contrôle. Avec ces mécanismes, en pratique, une application peut démarrer un plugin, mais dans un nouveau domaine de sécurité. La compilation est possible, mais le lancement d'une nouvelle application est fait dans ce nouveau domaine de sécurité.

4.7 Conclusion du chapitre

Notre modèle de calcul cherche à garantir des propriétés de sécurité spécifiques pour exécuter des programmes de manière sécurisée. L'*intégrité du flot de contrôle* (*CFI*) est une propriété critique permettant de se prémunir contre un attaquant souhaitant modifier le flot de contrôle d'un programme. Pour que cette propriété soit vérifiée, nous avons montré que ce système doit être capable d'extraire le *CFG* du programme, ce qui n'est pas possible en général à cause de l'existence des *prédicats à trappe*. Nous définissons un modèle de calcul pour réaliser une extraction facile et précise d'un *CFG* d'un programme. Nous comparons trois *ISAs* : une avec des sauts indirects (ISAv1), une sans sauts indirects (ISAv2) et une avec des restrictions sur les sauts indirects (ISAv3). Nous montrons qu'un système de chiffrement *AES* peut être réalisé dans l'ISAv3 sans perte de performance tout en permettant l'extraction du *CFG*. Nous montrons aussi qu'un schéma de *distributeur* (*dispatcher*) nécessite des sauts vers des adresses valides (*forward indirect jump*). Nous observons que ces *ISAs* peuvent entraîner une baisse de performance lorsqu'un programme possède un nombre important de sorties. Nous avons aussi montré que la

propriété de *CFI* ne peut pas être totalement garantie sans des contraintes sur les transferts de mémoires indirects, à cause de l'émergence de *machines virtuelles*. Nous allons voir dans les chapitres suivants que l'analyse d'un *CFG* à l'aide une théorie nécessite un certain nombre de conditions.

Chapitre 5

Notre approche sur la détection de prédicats opaques

Sommaire

5.1 Introduction 103
5.2 Définitions et propriétés 106
5.3 Détection de structure indépendante 118
5.4 Évaluation et Expériences 123
5.5 Discussions 126
5.6 Conclusion du chapitre 127

Résumé Nous proposons une nouvelle technique de détection des prédicats opaques, lorsqu'un moteur *DSE* analyse un programme. Cette détection se base sur une propriété de la théorie du premier ordre utilisée pour interpréter un programme qui contient des prédicats opaques. Nous évaluons notre technique avec quelques programmes contenant des prédicats opaques connus. Ces travaux ont été réalisés conjointement avec Olivier DECOURBE (Inria), Sébastien JOSSE (Dga), et Caroline FONTAINE (CNRS).

5.1 Introduction

Un prédicat définit une expression pouvant retourner les valeurs de vérité *vrai* (*true*) ou *faux* (*false*). L'opacité d'un prédicat peut rendre difficile son interprétation par un outil d'analyse automatique. Son exécution est facile, mais sa détection non. Les prédicats opaques peuvent être utilisés comme une technique de protection logicielle pour rendre plus difficile la reconstruction du *graphe de flot de contrôle* (*control flow graph* (*CFG*)) d'un programme exécutable. Dans ce cas, l'utilisation d'un prédicat opaque permet une protection contre une analyse statique, réalisée par exemple par un *moteur d'exécution symbolique dynamique*. Dans cette situation, le

programme est représenté sous forme de contraintes. La résolution de ces contraintes permet de connaître le comportement du programme. Les prédicats opaques allongent le temps nécessaire pour résoudre ces contraintes. Malheureusement, les prédicats opaques apparaissent sous différentes formes, et sont rarement utilisés seuls, comme en témoignent les nombreux travaux à ce sujet [CTL97 ; CTL98 ; DP08 ; SI14 ; Sch+16 ; BP18 ; Xu+18]. La détection des prédicats opaques apparaît donc comme une tâche difficile.

5.1.1 Problème

Comment peut-on améliorer l'interprétation des prédicats opaques pour les détecter de manière générique ? Les prédicats opaques sont populaires, car ils sont peu coûteux. Ils sont de surcroît faciles à composer avec d'autres techniques d'obfuscation (les formules *MBA* [1], ou les fonctions de hachages). Les moteurs d'*exécution symbolique dynamique* (*DSE*) utilisent des solveurs *SMT*. Ces solveurs sont de puissants outils permettant de connaître la satisfiabilité d'un ensemble de contraintes, donnant la possibilité de retourner un modèle qui est une solution pour ces contraintes. Depuis un code source, ou depuis des traces d'exécutions, si un prédicat opaque est présent dans une requête, le temps de résolution de cette requête est anormalement long. Ce surcoût temporel a pour but de décourager une analyse automatique, et de protéger les informations contenues dans le programme après l'introduction de ce prédicat. Pour éviter ce surcoût temporel, différentes méthodes avancées de détection existent en dehors de la résolution de la requête, mais aucune méthode à notre connaissance ne propose d'intervenir pendant la résolution de la requête contenant le prédicat opaque.

5.1.2 Objectifs et contributions

Comme nous l'avons vu dans le chapitre 2, la déobfuscation de programmes protégés par des prédicats opaques n'est possible que si le prédicat opaque est au moins détecté. Cette détection peut être réalisée par la détection d'une propriété du prédicat (par exemple : l'invariance du comportement du prédicat [Min+15]), ou de l'une de ces conséquences (par exemple : l'utilisation d'un code inutile sur la branche non utilisée [TH20]). Ici, nous nous plaçons du côté de l'attaquant en suivant un scénario d'*homme à la fin* (*MATE*) (*cf.* sous-section 2.2.2) : l'attaquant n'a pas accès au code source ni au binaire non-protégé, mais il possède un accès total au programme final protégé. Notre but est de détecter et d'analyser un programme contenant

1. Les formules *Mixed Boolean Arithmetic* (*MBA*) sont des expressions définies à l'aide des opérations arithmétiques classiques ($+, -, \times, /$) et des opérateurs booléens ($\oplus, \wedge, \vee, \neg$) [Eyr17].

des prédicats opaques. Nous conjecturons que la détection de prédicats opaques est possible en améliorant les moteurs d'*exécution symbolique dynamique* existants, sans apporter de nouveaux modules évolués. L'analyse d'un programme contenant un prédicat opaque par un moteur *DSE*, oblige ce moteur à fabriquer une requête *SMT*. Cette requête est résolue à l'aide d'un solveur *SMT*. Pour résoudre cette requête, le solveur *SMT* fera appel à un solveur *SAT*, pour résoudre des formules propositionnelles. C'est ce que nous appelons les résolutions intermédiaires de la requête *SMT*. Dans ce chapitre, nous regardons le taux d'évolution du temps de ces résolutions intermédiaires. Si un certain seuil est dépassé, nous conjecturons que nous venons de détecter des éléments d'un prédicat opaque. À partir du moment où ces éléments sont détectés, nous retournons à l'utilisateur l'endroit où le prédicat est détecté et nous laissons à l'utilisateur le soin de modifier le code. Nous détaillons maintenant les différents points que nous apportons :

- nous proposons un scénario pouvant expliquer l'origine des prédicats opaques, en nous appuyant sur la présence d'une structure indépendante dans le prédicat ;

- nous proposons la mise en place d'un distingueur au niveau des appels de résolution pour détecter les prédicats opaques ;

- à notre connaissance, notre approche est la première permettant de comprendre l'origine des effets d'opacité des prédicats opaques lors d'une analyse statique ;

- nos évaluations ont été réalisées à partir de différents jeux de programmes disponibles. Les résultats expérimentaux montrent l'efficacité de notre méthode de détection.

L'identification des prédicats opaques à l'aide d'un moteur *DSE* n'est pas nouvelle [Min+15 ; BDM17]. Les moteurs *DSE* seuls ne semblent pas adaptés pour des requêtes non satisfiables créées par des prédicats opaques, et doivent être adaptés pour y répondre, comme le montrent Bardin, David et Marion [BDM17]. Avec cette nouvelle méthode, nous souhaitons proposer à la communauté une contribution sur la compréhension de l'origine des prédicats opaques. Pour réaliser notre analyse, nous nous sommes appuyés sur la méthodologie proposée par [Sch+16 ; Oll+19a ; GF19].

Le reste du chapitre est organisé de la manière suivante. La section 5.2 présente le cadre général. La section 5.3 décrit le fonctionnement de notre approche et présente un exemple d'application. Dans la section 5.4 nous évaluons notre méthode. Une discussion sur notre méthode est proposée dans la section 5.5, suivie des pistes pour des travaux futurs.

5.1.3 Travaux associés

Les premiers travaux sur la détection des prédicats opaques se sont limités aux *prédicats opaques statiques* (*cf.* sous-section 2.3.1), à l'aide de la méthode de *filtrage par motifs* (*pattern matching*) [2]. Des travaux plus récents tentent de détecter automatiquement les prédicats connus et inconnus [Min+15 ; Rin17 ; BDM17 ; GF19 ; TH20]. L'utilisation d'un moteur d'*exécution symbolique dynamique* (*DSE*) permet par exemple de détecter une quantité restreinte d'expressions invariantes dans un programme [BDM17]. Malheureusement, l'utilisation de ces outils se heurte aux expressions évoluées, dont la résolution est difficile, voire impossible. Tofighi-Shirazi *et al.* [Tof+19] proposent une première solution à cette difficulté en utilisant une approche de détection par de l'*apprentissage automatique* (*machine learning*), utilisant un modèle de classification par un *arbre de décision* (*decision tree*). Cette solution permet non seulement de détecter des prédicats opaques connus, mais également certains non rencontrés jusque là. Plus récemment, Tung et Harris [TH20] proposent une détection générique de prédicats opaques en observant plusieurs conséquences de la présence de prédicats opaques dans un programme, comme la présence de code inactif dans la branche qui n'est pas exécutée.

5.2 Définitions et propriétés

Dans cette section, nous nous plaçons à la jonction de différents domaines de recherche (*algèbre* (*universal algebra*), *théorie des modèles calculables* (*computable model theory*) et problème *SMT*) afin de comprendre le fonctionnement de la résolution d'une requête définie dans une théorie du premier ordre, et contenant un prédicat opaque. Nous commençons par définir ce qu'est une structure. Nous expliquons ensuite les liens existant entre le problème d'isomorphisme et le problème *SMT*. Nous introduisons la différence entre *structure calculable*, *structure décidable*, et *structure automatique*, pour comprendre comment fonctionne la résolution d'une requête. Enfin, nous proposons d'aborder quelques propriétés des structures qui donnent lieu à l'allongement du temps de résolution d'une requête.

2. Le *pattern matching* est une méthode de vérification de la présence de constituants d'un motif.

5.2.1 Structure

Une *structure* $\mathfrak{A} := (\mathsf{A}; \sigma; \rho)$ est définie à l'aide d'un ensemble A, de fonctions [3] finies σ et d'une collection de relations [4] ρ définies dans cet ensemble. L'ensemble A est appelé *ensemble de définition* ou *domaine*. Ainsi, une structure peut être confondue soit avec ses collections de relations et de fonctions, soit avec son ensemble de définitions. En fonction de la branche des mathématiques choisie, la définition d'une structure peut prendre un sens différent. C'est pourquoi, nous utilisons deux définitions du concept de structures, que nous différencions par un suffixe : *structure-logique* et *structure-Bourbaki*.

5.2.1.1 Structure-logique

Les *structures-logiques* représentent les objets utilisés pour définir la sémantique d'un langage du premier ordre. Une *structure-logique* $\mathcal{A}$ est définie à l'aide d'un triplet (A, σ, I), représentant respectivement : un *ensemble de définitions* (ou *domaine*) A, une *signature* σ et une *interprétation* I de la *signature* dans ce *domaine*. Une *signature* est une collection de symboles de *fonctions* $\mathbf{f}$ et de symboles de *relations* $\mathbf{r}$, avec une fonction qui associe à chaque symbole une arité [5]. Une *interprétation* I est définie à l'aide d'un ensemble non vide $\mathcal{U}$, telle que chaque nom d'objet mentionné dans le langage est associé à un élément de $\mathcal{U}$. Un symbole de fonction $\mathbf{f}$ d'arité 0 est un symbole de *constante*. Une *théorie* est définie pour un langage formel donné, à l'aide d'un *domaine* et d'une *signature*. L'ensemble $\mathcal{U}$, ou l'interprétation dont il fait partie, forme un *modèle* d'une théorie lorsque tous les axiomes de cette théorie sont vrais relativement à cette interprétation. En d'autres termes, une *structure-logique* définie dans une *théorie* est un *modèle*. L'exemple 11 est une illustration d'une théorie définie dans un langage.
Ajoutons qu'une *structure-logique* relationnelle [6] est *homogène* (*homomogeneous*) si elle est dénombrable, que chaque sous-structure finie est *élémentairement équivalente* [7] et s'étend par un *automorphisme* [Mac11]. Autrement dit, une *structure-logique* est *homogène*, si elle possède une méthode d'élimination des quantificateurs. Une théorie *dénombrable* est *complète* si tous

3. Un symbole de fonction représente une expression contenant un ensemble de symboles d'opérations définies dans A^n, avec n un entier positif.

4. Une relation entre n éléments est définie sur un ensemble A, comme une partie R de A^n ($n = 2$ pour les relations binaires). Notons que si l'uplet $\bar{a} := (a_1, a_2)$ extrait de A^2 appartient à R, notée $\bar{a} \in R$, alors l'uplet $\bar{a}$ satisfait la relation R dans A, notée $\mathsf{A} \vDash R(\bar{a})$, sinon $\bar{a}$ ne satisfait pas R dans A, notée $\mathsf{A} \nvDash R(\bar{a})$.

5. L'arité d'un symbole est le nombre entier positif définissant le nombre d'arguments que le symbole accepte.

6. Une *structure-logique* composée uniquement de relations, et ne possédant pas de symbole de fonction, est dite *relationnelle*.

7. Deux structures sont *élémentairement équivalentes* lorsque leurs théories du premier ordre sont les mêmes.

ses modèles sont décidables [8]. Une théorie *dénombrable* et *complète* est *homogène* si tous ses modèles sont homogènes [9]. Par exemple, la *Théorie sans quantificateurs des vecteurs de bits de taille fixe* (*Closed quantifier-free formulas over the theory of fixed-size bitvectors*), notée *QF-BV*, est une théorie dénombrable, complète et homogène.

Exemple 11:
Pour le langage L:= $(=, \{0, +\})$, constitué d'un symbole de constante noté 0, d'un symbole de fonction binaire noté $+$ et d'un symbole de relation noté $=$, un exemple de structure peut être $\langle \mathbb{N}, 0, + \rangle$, ou $\langle \mathbb{Z}, 0, + \rangle$. Un exemple de théorie définie à l'aide de ce langage peut être la *Théorie des ensembles infinis dans le langage réduit à l'égalité* : $\exists x_1 x_2 \ldots x_n \wedge_{i \neq j} (x_i \neq x_j)$, pour tout entier $n > 0$.

5.2.1.2 Structure-Bourbaki

Les *structures-Bourbaki* définissent une classe de représentations « artificielles » [Bou70] permettant la simplification de la présentation et de la manipulation des concepts mathématiques exprimés dans les principales branches des mathématiques. Une *structure-Bourbaki* est une *structure-logique* définie à l'aide d'un langage du premier ordre, et des symboles de prédicats d'*égalité*, noté $=$, et d'*appartenance*, noté $\in$ [10]. Dans cette classe, nous trouvons par exemple les structures algébriques (par exemple : monoïde, groupe, espace vectoriel, *etc.*) ou les structures d'ordre (par exemple : arbre, treillis, *etc.*). L'un des éléments importants de cette classe se dénomme *automorphisme*. Les *automorphismes* permettent d'exprimer les symétries d'un objet mathématique $\mathfrak{X}$, en réalisant une *correspondance biunivoque* [11] (*one-to-one correspondence*) entre les différents éléments qui composent cet objet. L'ensemble des *automorphismes* d'un objet $\mathfrak{X}$ est noté $\mathtt{Aut}(\mathfrak{X})$. La composition des relations de cet ensemble possède une *structure-Bourbaki* de groupe [12], telle que : l'élément neutre de ce groupe est la fonction identité [13],

8. Un modèle $\mathcal{A}$ est *décidable* si la théorie $\mathrm{Th}(\mathcal{A}, a)_{a \in \mathcal{A}}$ (la théorie définie par l'ensemble des paires de formules du premier ordre ϕ, associée à une séquence finie $\bar{a}$ du domaine de $\mathcal{A}$) est calculable [Mor76].

9. Une structure homogène interprétée dans une théorie est un modèle homogène.

10. Le symbole $\in$ est défini dans la Théorie des ensembles moderne (ou Théorie des ensembles de Zermelo-Fraenkel avec axiome du choix (ZFC)).

11. Qui fait correspondre un élément d'un ensemble à un autre élément de l'autre ensemble.

12. Un groupe est un ensemble muni d'une opération binaire qui satisfait les quatre axiomes suivant : loi de composition interne, associativité, élément neutre, éléments symétriques.

13. La fonction identité est la fonction qui laisse invariants les éléments d'un ensemble.

et l'inverse d'un automorphisme est sa réciproque [14]. Le groupe des automorphismes possède deux sous-groupes normaux [15] : le groupe des *automorphismes intérieurs*, noté $\texttt{Inn}(\mathfrak{X})$, et le groupe des *automorphismes extérieurs*, noté $\texttt{Out}(\mathfrak{X})$. La composition d'éléments de $\texttt{Inn}(\mathfrak{X})$ permet de conserver les *isomorphismes* d'une structure, c'est-à-dire qui ne modifie pas les relations définies dans une structure. La composition des éléments de $\texttt{Out}(\mathfrak{X})$ ne permet pas de conserver les *isomorphismes* d'une structure, c'est-à-dire que la composition d'une structure avec des éléments de $\texttt{Out}(\mathfrak{X})$ ne permet pas de conserver les relations définies dans la structure. L'exemple 12 présente un cas de conservation d'une opération de transposition.

Exemple 12:
Soit un ensemble de trois éléments E=$\{a, b, c\}$, les transpositions [a] $(a\,b), (a\,c), Id$ [b] engendrent le groupe symétrique S_3 [c]. Soit la fonction ϕ qui envoie une opération de transposition sur une opération de transposition, alors $\phi \in \texttt{Aut}(S_3)$, et $\phi \in \texttt{Inn}(S_3)$, car tous les cycles présents dans S_3 sont préservés par ϕ.

a. Une transposition sur un ensemble est une permutation qui échange la position de deux éléments, et laisse toutes les autres fixes .
b. Id : transposition d'identité, c'est-à-dire qui laisse invariant la position de deux éléments.
c. Le groupe symétrique S_3 est le groupe des permutations d'un ensemble à 3 éléments.

Le groupe $\texttt{Out}(\mathfrak{X})$ est défini par le groupe quotient $\texttt{Aut}(\mathfrak{X})/\texttt{Inn}(\mathfrak{X})$. Si le groupe $\texttt{Out}(\mathfrak{X})$ correspond au groupe identité, alors le groupe $\texttt{Aut}(\mathfrak{X})$ correspond exactement au groupe $\texttt{Inn}(\mathfrak{X})$. L'exemple 13 présente un cas où un groupe d'automorphisme extérieur existe.

Exemple 13:
Soit un ensemble de trois éléments E=$\{a, b, c\}$, les permutations suivantes $(a\,b\,c), (a\,c\,b), Id$ engendrent le groupe alterné A_3, un sous-groupe normal du groupe symétrique S_3. L'automorphisme du groupe S_3 permettant de passer de $(a\,b\,c)$ à $(a\,c\,b)$ est un *automorphisme intérieur*, car $(a\,c\,b) = (a\,b)(a\,b\,c)(a\,b)$. Mais cet automorphisme est extérieur dans A_3, car il n'existe pas d'élément h dans A_3 préservant les cycles, c'est-à-dire tel que $(a\,c\,b) = h^{-1}\,(a\,b\,c)\,h$.

14. Si R est une relation binaire de E sur F (notée $R \in E \times F$), la relation R^{-1} de F sur E, (notée $R^{-1} \in F \times E$), est appelée *relation inverse de* R telle que : $R^{-1} := \{(x, y) \in F \times E \,|\, (y, x) \in R\}$

15. Un sous-groupe N d'un groupe G est *normal* dans G si et seulement si $gng^{-1} \in N$ pour tout $g \in G$ et $n \in N$. En d'autres mots, un sous-groupe N d'un groupe G est invariant par la conjugaison par des membres du groupe G.

5.2.2 Le problème d'isomorphisme et d'équivalence

Le *problème d'isomorphisme* (*isomorphism problem*) consiste à répondre à la question de savoir si deux structures sont isomorphes. Comme nous allons le voir dans la suite de ce chapitre, ce problème est important dans la résolution de requêtes *SMT*. Khoussainov et Minnes [KM07] proposent une réponse formelle et simple à ce problème à l'aide d'invariants permettant de décrire chaque structure, et en vérifiant si un *isomorphisme* est présent. Par exemple, en décrivant chaque structure dans un espace vectoriel utilisant des vecteurs de taille fixe. Malheureusement, l'étude de certaines classes de structures (par exemple : les groupes abéliens [16]) montre qu'il est impossible de fixer des invariants d'*isomorphisme*. Ainsi, en général, il n'existe pas de solution pour le *problème d'isomorphisme*. Une autre façon de répondre à ce problème est d'essayer de trouver une approximation pour comparer deux structures. Indépendamment de toutes logiques, une manière simple d'approximer peut être définie par une *équivalence élémentaire* (*elementary equivalence*) [17] pour un langage du premier ordre. Une des instances du *problème d'équivalence* consiste à vérifier qu'une structure, représentée par un ensemble de formules définies dans une théorie du premier ordre, soit satisfiable facilement, dit autrement cela revient à essayer de résoudre le problème *SMT* (*cf.* section 3.3).

5.2.3 Structure calculable, décidable ou automatique

L'utilisation des structures en informatiques permet de classer les structures en trois familles distinctes : les structures calculables, les structures décidables, et les structures automatiques. L'interprétation automatique d'un programme revient à trouver un équilibre entre ces trois structures.

5.2.3.1 Structure calculable

Les structures *calculables* correspondent aux structures dont le domaine de définition peut être représenté par un ensemble accepté par une machine de Turing, et toutes les relations et les fonctions de ces structures sont aussi acceptées. Cenzer et Remmel [CR91] démontrent que les structures calculables suivantes sont isomorphes d'un point de vue de la calculabilité (*computable isomorphic*) à une *structure calculable en temps polynomial* : toutes les structures

16. Un groupe abélien, ou un groupe commutatif, est un groupe possédant une loi de composition interne commutative.

17. Pour rappel : deux structures sont *élémentairement équivalentes* lorsque leurs théories du premier ordre sont les mêmes.

relationnelles, toutes les algèbres Booléennes, et les structures arithmétiques définies sur l'ensemble des entiers naturels. Dans le même papier, les auteurs démontrent aussi que certaines structures calculables ne peuvent pas être isomorphes à une *structure calculable en temps polynomial* comme les groupes abéliens, ou les structures relationnelles dont le domaine de définition est borné. En pratique, comme trouver un *isomorphisme* est une tâche difficile, c'est une *équivalence élémentaire* qui est utilisée [18]. Plus précisément, un programme exprimant des structures de la première catégorie peut être exécutés en temps polynomial, et ceux de la deuxième catégorie non. Ces résultats peuvent être mis en parallèle du comportement des prédicats opaques qui sont facilement calculables, mais qui génèrent une longue séquence de modèles. Certains prédicats opaques intègrent la définition de la première catégorie de structure, ce qui peut expliquer pourquoi leurs exécutions s'effectuent facilement.

5.2.3.2 Structure décidable

Nous pouvons définir qu'une structure $\mathfrak{A}$ est *décidable* de différentes manières. La plus simple est de dire qu'il existe un algorithme permettant de décider précisément quelle phrase est vraie dans cette structure. Si cet algorithme est automatisable, c'est-à-dire qu'il peut être reconnu par un automate fini, alors cet algorithme résout une instance du problème *SMT*.

5.2.3.3 Structure automatique

Dans le but de mieux capturer les invariants du *problème d'isomorphisme* donné par Cenzer et Remmel [CR91], Khoussainov et Nerode [KN94] proposent de restreindre le modèle de calcul d'une structure calculable pour obtenir une nouvelle classe de structures appelée *structures automatiques* (ou parfois aussi appelée *structure automatique de chaînes de caractères* (*string automatic structures*) [BG00]). Ce modèle de calcul est basé sur les automates finis. Les méthodes de détections et de vérifications d'erreurs utilisent des *structures automatiques*, comme le montrent les méthodes d'*analyse statique* (*cf.* sous-section 3.1.1) ou les méthodes de *vérification des modèles* (*model checking*) [19]. L'exemple 14 présente différents cas de *structures automatiques*.

18. Le calcul d'une *équivalence élémentaire* est réalisé par une *méthode de va-et-vient* (*back and forth method*).

19. La *vérification des modèles* (*model checking*) est une méthode formelle utilisée pour vérifier l'*exactitude* (*correctness*) des propriétés satisfaites d'un programme informatique [BK08].

Exemple 14:
Un exemple de structure automatique peut être la structure de mots suivante : $(\{0,1\}^*, L, R, E, \preceq)$, telle que pour des mots binaires $x, y \in \{0,1\}^*$, $L(x) = x0$, $R(x) = x1$, la relation $E(x,y)$ est vérifiée si et seulement si $|x| = |y|$, et $\preceq$ est l'ordre lexicographique.
Un autre exemple peut être le *graphe de flot de contrôle* d'un programme.
La structure suivante $(\{0,1\}^*, \wedge, \vee, \neg\}$ est automatique, car les opérations sur des chaînes binaires sont reconnaissables par un automate fini.

L'analyse d'un programme à l'aide d'un moteur d'*exécution symbolique dynamique* réalise une interprétation d'une structure. Une *structure automatique* est une *structure relationnelle* (*cf.* sous-section 5.2.1.1) dont le *domaine* et les *relations* sont reconnus par un automate fini lisant des mots finis. Les *structures automatiques* possèdent de bonnes propriétés de décidabilité et de définabilité . Elles représentent une sous-catégorie de la *classe des structures calculables et dénombrables* où « calculable » est remplacé par « reconnu par un automate fini ». Si de plus nous remplaçons « automate » par « automate d'arbres » ou par « automate de mots », nous obtenons respectivement les notions de structures automatiques d'arbres [BG00] et structures automatiques de mots [Hod82 ; KN94]. Comme les *structures automatiques*, ces structures sont closes par un langage du premier ordre et décidables par une théorie du premier ordre. Elles ont aussi la particularité d'être très facilement calculables.

5.2.4 Présentation automatique et résolution d'une requête

Des trois classes de structures que nous venons de décrire, nous pouvons maintenant faire le lien avec les moteurs d'*exécution symbolique dynamique*, et les solveurs *SMT*. En effet, lors de l'analyse d'un programme, un moteur *DSE* fait appel à un solveur *SMT*, après avoir traduit un prédicat de chemin en une requête. Autrement dit, le moteur réalise une *équivalence élémentaire* entre une structure calculable et une structure décidable dans une théorie du premier ordre fixée, et *vérifie* (*check*) ensuite l'existence d'un *isomorphisme* entre la requête et la théorie choisie [Imm12] à l'aide d'un solveur. Pour que cette résolution soit utilisable dans la vie réelle, la résolution d'une requête doit tendre vers un temps polynomial, c'est-à-dire que la structure décidable doit être automatique. En pratique, cela revient à interpréter la structure décidable par une *présentation automatique*, ce que nous allons voir maintenant.

5.2.4.1 Présentation automatique

Introduit par KHOUSSAINOV et NERODE [KN94] et IMMERMAN [Imm12], une *présentation automatique* permet de décrire les structures finies utilisées en informatique à l'aide d'une théorie du premier ordre. Par exemple, les méthodes de détections et de vérifications d'erreurs basées sur l'analyse symbolique définie par KING [Kin75] peuvent être vues comme une application concrète de l'utilisation des présentations automatiques. Une *présentation automatique* d'une structure relationnelle permet de réaliser une représentation abstraite des éléments d'une structure à l'aide d'un langage régulier, tel que les relations peuvent être reconnues par des automates synchrones à états finis. Ces automates interprètent en parallèle la lecture de plusieurs mots, en utilisant un *encodage de convolution* (*convolution encoding*) [KN94], comme le montre l'exemple 15. Plus simplement, une structure $\mathfrak{A}$ admet une *présentation automatique* si elle est isomorphe à une *structure automatique* $\mathfrak{B}$. Pour chaque structure, il existe un nombre fini de présentations automatiques. Une structure admettant une présentation automatique est alors définie comme une structure *AF-présentable* (*FA-presentable structure*). Chaque structure *AF-présentable* possède une théorie du premier ordre décidable.

Exemple 15:

Soit trois mots (bison, cable, halo). Pour lire efficacement ces mots par un automate fini, l'automate réalise un encodage de convolution en lisant en une seule fois la $n^{ème}$ lettre de chaque mot. Pour ce faire, nous pouvons commencer par aligner verticalement les différentes lettres de chaque mot, en ajoutant un *symbole de rembourrage* (*padding symbol*) $\square$, non défini dans l'alphabet d'origine, pour les espaces manquants. Si nous reprenons nos trois mots, nous obtenons :

b i s o n
c a b l e
h a l o $\square$

Puis, nous associons ensemble les $n^{ème}$ lettres de ces mots, pour obtenir des triplets de lettres.

$$(\mathsf{b},\mathsf{c},\mathsf{h})\ (\mathsf{i},\mathsf{a},\mathsf{a})\ (\mathsf{s},\mathsf{b},\mathsf{l})\ (\mathsf{o},\mathsf{l},\mathsf{o})\ (\mathsf{n},\mathsf{e},\square) \tag{5.1}$$

Ce qui nous permet d'obtenir un nouvel alphabet :

$$A^3_\square = ((A \cup \{\square\}) \times (A \cup \{\square\}) \times (A \cup \{\square\}))/\{(\square,\square,\square)\}$$

Ainsi, la nouvelle forme de l'exemple est un mot écrit à partir de l'alphabet $A^3_\square$. Un langage de mots comme (bison, cable, halo) est accepté par un automate fini si l'ensemble des nouvelles

formes de ces mots est accepté par un automate fini défini sur $A_{\square}^{3}$.

Les travaux de KUSKE et WEIDNER [KW11] montrent que toutes les *structures automatiques d'arbres* (*cf.* sous-section 5.2.3) admettent une présentation automatique d'arbre injective. Cette présentation est définie dans la classe de complexité descriptive *PSPACE* (*polynomial space*) [20] et sa taille est exponentielle. Ainsi, nous pouvons dire que l'analyse d'un programme à l'aide d'un moteur *DSE* est une présentation automatique injective réalisée à l'aide d'un solveur *SMT*. Malheureusement, à notre connaissance et comme le suggèrent aussi KUSKE et WEIDNER [KW11], aucune démonstration formelle ne permet de savoir si l'explosion exponentielle apparaît lorsqu'une présentation automatique de mots est traduite dans une présentation automatique d'arbre injective. Vu autrement, dans le cas d'une analyse de programme à l'aide d'un moteur *DSE*, il n'existe pas de preuve formelle permettant de dire quand l'explosion exponentielle d'une analyse apparaît. Ainsi, il n'existe pas de preuve formelle permettant de dire quand un prédicat opaque peut être utilisé comme une méthode de protection logicielle.

5.2.4.2 Résolution d'une requête

Nous expliquons maintenant comment une requête est résolue, d'un point de vue ensembliste. Nous nous appuyons sur les travaux de BARBINA et MACPHERSON [BM07], qui expliquent comment reconstruire une structure homogène. À notre connaissance, aucun travail actuellement ne met en lien cette résolution avec un moteur d'*exécution symbolique dynamique*. Nous souhaiterions éclaircir ce point pour mieux comprendre les conditions d'apparition des prédicats opaques.

5.2.4.2.1 Ensembles minimaux et automorphismes Comme le montre IMMERMAN [Imm12], la résolution d'une requête définie dans une théorie complète, consiste à vérifier l'existence d'un *isomorphisme* entre la requête et les axiomes de la théorie. Comme cet *isomorphisme* n'est pas toujours possible, une approximation est calculée par une *équivalence élémentaire*. Cette *équivalence élémentaire* peut prendre la forme d'une *AF-présentation* ou bien d'une séquence de présentations [21]. Lorsque la résolution de la requête est calculable et se finie, si la structure construite pour représenter cet *isomorphisme* est un modèle, alors la requête est dite satisfiable (*sat*), sinon, cette structure n'est pas un modèle et la requête est dite non-satisfiable (*unsat*). Nous

20. `https://complexityzoo.uwaterloo.ca/Complexity_Zoo`

21. Une séquence de présentation est la représentation de l'utilisation d'une *méthode de va-et-vient* (*back and forth method*).

avons noté, dans le chapitre 3, que les théories utilisées par les solveurs *SMT* sont homogènes [22]. L'utilisation de ces théories permet à la fois de résoudre dans la théorie un certain nombre de contraintes, mais aussi de traduire les formules *SMT* en formules atomiques résolubles à l'aide d'un solveur SAT. Les atomes de ces formules sont une représentation des éléments définis dans l'*ensemble fortement minimal* (*strongly minimal set*) [23] de la structure, ou plus simplement dans l'ensemble des *éléments irréductibles* (*irreducible elements*) [HW+93]. Ce qui nous permet de comprendre que lors de la résolution d'une requête *SMT*, la construction d'un modèle s'effectue à partir de l'ensemble des éléments irréductibles de la structure contenue dans la requête. Cet ensemble se prolonge par une *équivalence élémentaire* avec les éléments définis dans l'automorphisme de la structure. Plus précisément, l'ensemble se prolonge par les éléments définis dans l'automorphisme de la théorie qui sont isomorphes ou élémentairement équivalents aux éléments de l'automorphisme de la structure reconstruite [BM07].
Nous montrons dans l'exemple 16 les éléments de la théorie QF-BV.

Exemple 16:
Soit QF-BV, la *théorie sans quantificateurs des vecteurs de bits de taille fixe*. Le groupe d'automorphisme de la théorie QF-BV est le *groupe linéaire affine* (*affine linear group*) AGL($\mathbb{F}_2$,k) [a], isomorphe au produit semi-direct extérieur de l'espace vectoriel $\mathbb{K}^n$ [b] par le *groupe linéaire* (*linear group*) GL(n,k), agissant par transformations linéaires [BKS15]. Les éléments de AGL($\mathbb{F}_2$,k) sont 3-transitifs, c'est-à-dire qu'ils peuvent être représentés au format *CNF* par 4 conjonctions. Ainsi, la résolution d'une requête définie dans la théorie QF-BV s'effectue par la translation d'éléments 3-transitifs appartenant au groupe AGL($\mathbb{F}_2$,k).

a. AGL($\mathbb{F}_2$,k) : le groupe linéaire affine dont les éléments sont encodés dans des vecteurs de taille k et les éléments appartiennent au corps fini de deux éléments $\mathbb{F}_2$.
b. $\mathbb{K}^n$: un espace vectoriel de taille n, définie sur un corps $\mathbb{K}$.

5.2.4.2.2 Automorphisme intérieur et extérieur Malheureusement, les éléments constituant le(s) groupe(s) des automorphismes d'une théorie n'ont pas tous les mêmes propriétés. En effet, la construction d'une structure à partir de ces éléments peut se retrouver dans deux situations bien distinctes. La première situation concerne les éléments qui ne modifient pas les relations de la structure existante, c'est-à-dire qui ne créent pas de conflit dans la démarche de déduction.

22. Pour rappel : les théories homogènes sont les théories dont les *équivalences élémentaires* se prolongent dans les automorphismes.

23. Un sous-ensemble d'une structure défini comme l'ensemble des réalisations d'une formule ϕ est appelé un *ensemble minimal* si chaque sous-ensemble de son domaine est soit fini, soit co-fini. Si cet ensemble reste vrai pour toutes les équivalences élémentaires, alors cet ensemble est appelé l'*ensemble fortement minimal*.

Ces éléments appartiennent à l'automorphisme intérieur de la structure. La seconde situation concerne les éléments qui ne conservent pas les relations de la structure existante, c'est-à-dire qui créent au moins un conflit de résolution. Ces éléments appartiennent à l'automorphisme extérieur de la structure. Nous supposons que peu importe les méthodes de déductions utilisées, les nouveaux éléments créés par ces méthodes (par exemple les *variables fraîches* (*fresh variables*)) aidant à la construction de structures permettant la résolution d'une requête, sont définis dans le groupe de l'automorphisme extérieur de la structure construite dans au moins deux situations. Dans le premier cas, si l'automorphisme extérieur de la théorie est trivial [24], chaque élément non neutre capturé par le système de déduction appartient toujours à l'automorphisme intérieur de la théorie, mais peut être non isomorphe à l'un des éléments de l'automorphisme intérieur de la structure construite. Dans le deuxième cas, l'automorphisme extérieur de la théorie n'est pas trivial, alors un élément capturé peut être isomorphe à un élément appartenant au groupe d'automorphisme extérieur de la structure construite. En pratique, les éléments appartenant à l'automorphisme extérieur de la structure sont naturellement présents dans les clauses de conflits ou la *justification* [25] générés dans ou par les solveurs *SMT*. À notre connaissance, une structure constituée uniquement d'éléments définis dans l'automorphisme extérieur d'une structure peut être élémentairement équivalente à une structure constituée d'éléments définis dans l'automorphisme intérieur de cette structure, si tous les éléments de l'automorphisme extérieur ont été exhibés. L'exemple 17 illustre une situation où le temps de résolution d'une requête *SMT* définie dans la théorie QF-BV peut devenir long.

Exemple 17:
Les groupes des automorphismes de AGL et GL sont équivalents, dans le sens où ils possèdent le même groupe d'automorphisme [BKS15]. Ainsi, lorsqu'une structure définie dans l'*espace vectoriel de dimension infinie dénombrable sur le corps à deux éléments* (*countably infinite dimensional vector space over the two-element field*) noté $\mathbb{F}_2^k$, est interprétée dans la théorie QF-BV, il existe une équivalence élémentaire entre la représentation dans $\mathbb{F}_2^k$ et la représentation dans QF-BV. De plus, comme la procédure de décision principale de la théorie QF-BV est équivalente à la technique du *bit-blasting*, nous pouvons constater que c'est une autre équivalence élémentaire utilisée pour obtenir une procédure de décision en temps polynomial. D'un autre point de vue, voici ce que nous pouvons dire. Soit une structure $\mathfrak{S}$ définit dans $\mathbb{F}_2^k$. $\mathfrak{S}$ est calculable par une machine réelle. $\mathfrak{S}$ est interprétable dans la théorie QF-BV, qui est une théorie homogène. Alors il existe au moins une équivalence élémentaire entre la structure $\mathfrak{S}$

24. Trivial dans le sens où le groupe contient un seul élément : l'élément neutre.
25. La *justification* est le sous-ensemble de clauses de conflits qui sont non satisfaites, noté *UC*

et sa théorie. Cette équivalence élémentaire peut être calculée à l'aide d'un solveur *SMT*. Le temps réel du calcul réalisant cette équivalence va dépendre de la nature des éléments capturés par le solveur *SMT*. Par exemple, si une structure est uniquement composée d'éléments définis dans l'automorphisme extérieur de $\mathbb{F}_2^k$, comme dans le cas d'un groupe abélien [HW+93], le temps de résolution d'une requête *SMT* sera le plus long possible.

5.2.5 Croissance de recouvrement

Dans la situation où une structure est composée de relations localement finies et régulières, cette structure est automatique, mais pas *AF-presentable* [Cai+10].Dans ce cas, l'interprétation de cette structure dans une théorie dénombrable, complète et homogène, donne une suite bornée d'éléments finis [Kho+04 ; HW+93], appelée parfois suite de recouvrement. La croissance [26] de cette suite permet de connaître certaines propriétés de la structure, comme le rappel l'étude de Macpherson [Mac11]. Par exemple, pour une structure-Bourbaki de groupe G virtuellement [27] abélien et :

- automatique : si le groupe est engendré par une partie [OT05]
- decidable : si le groupe est virtuellement résoluble [Rom81]

Alors, la croissance de la suite de recouvrement de G est :

- polynomiale : si son n^{eme} générateur de groupe est de taille polynomiale n.
- exponentielle : lorsque le nombre de symboles acceptés d'une relation d'arité k est plus grand que le nombre de symboles n utilisés pour interpréter cette relation [Cam83].

Il n'existe pas de borne maximale pour le taux de croissance [Cam09]. Enfin, ajoutons qu'une croissance exponentielle de recouvrement d'une structure dans une *théorie homogène* peut induire la propriété d'indépendance (définition 5.1) définie de la manière informelle suivante : chaque union de chaque élément de la suite de recouvrement peut être un modèle dans cette théorie. Une structure possédant la propriété d'indépendance est appelée *structure indépendante*. L'exemple 18 illustre le cas d'une structure de groupe dont la croissance de recouvrement est polynomiale. Cet exemple sera réutilisé dans le chapitre suivant.

26. Pour comparer le taux de croissance de deux suites non décroissantes de termes positifs $\{a_n\}$ et $\{b_n\}$, il faut évaluer la relation suivante : $\lim_{n\to\infty} \frac{a_n}{b_n}$. Si le résultat est 0 alors la suite $\{b_n\}$ croit plus vite que $\{a_n\}$, si le résultat est ∞ alors la suite $\{a_n\}$ croit plus vite que $\{b_n\}$.

27. Un groupe G possède virtuellement une propriété si un sous-groupe de G possède cette propriété.

Définition 5.1 *Soit $\mathcal{T}$ une théorie complète, alors une structure relationnelle ϕ possède une propriété d'indépendance si pour chaque modèle $\mathcal{M}$ de $\mathcal{T}$, il existe, pour chaque $n \in \mathbb{N}$ fini, une famille d'uplets $\bar{b}$, tel que pour chacun des 2^n sous-ensembles X de n, il existe un uplet $\bar{a}$ dans $\mathcal{M}$ tel que : $\mathcal{M} \models \phi(a, b_i) \Leftrightarrow i \in X$*

Exemple 18:
Soit le groupe 4-Klein [a] noté G. G est un groupe abélien, isomorphe au produit direct du groupe d'ordre 2 par lui-même ($\mathbb{Z}/2\mathbb{Z} \times \mathbb{Z}/2\mathbb{Z}$), qui n'est pas cyclique. Soit la relation suivante définie dans QF-BV, construite à partir d'un générateur d'un groupe de 4-Klein : $r'_0 : x^2 + 25 \bmod 4$, où x est une variable de 32-bits. Alors r'_0 est une relation localement finie et régulière, qui contient une structure automatique. Comme les générateurs du groupe G sont de taille constante, la croissance de la séquence de recouvrement est polynomiale.

a. `https://groupprops.subwiki.org/wiki/Klein_four-group`

Bilan

Nous venons de voir qu'une *structure automatique* peut être interprétée dans une théorie pour former un ou plusieurs modèles. Cette interprétation est réalisée en pratique par une *équivalence élémentaire*, à l'aide de théories homogènes. Ainsi, le temps de résolution d'une requête *SMT* dépend en partie de propriétés de la structure interprétée. La propriété d'indépendance peut être détectée par le taux de croissance de recouvrement.

5.3 Détection de structure indépendante

Des informations que nous venons de voir dans la section précédente, nous pouvons maintenant proposer notre conjecture sur la nature des prédicats opaques. Pour vérifier notre conjecture, nous proposons une méthode basée sur le taux de croissance de recouvrement. Ce distingueur est facile à mettre en œuvre dans un moteur d'*exécution symbolique dynamique* utilisant un solveur *SMT*.

5.3.1 Conjecture et justifications

Nous proposons la conjecture suivante pour les prédicats opaques :

L'interprétation d'un prédicat opaque dans une théorie homogène recouvre une structure automatique possédant la propriété d'indépendance.

Pour justifier notre conjecture, nous avons réalisé l'analyse suivante. L'utilisation des prédicats opaques en tant que technique d'obfuscation logicielle est une transformation sémantique de programme. Cette transformation permet de conserver quasiment les mêmes temps d'exécution que le programme original. Autrement dit, si P représente le programme de départ, et P' le nouveau programme contenant un prédicat opaque, P et P' sont des structures automatiques. De plus, une analyse d'un programme par un moteur *DSE* réalise une présentation automatique à l'aide d'un solveur *SMT*. Dit autrement, le programme P est *AF-présentable*. L'obfuscation d'un programme est considérée comme efficace si une analyse réalisée à l'aide d'un moteur *DSE* nécessite plus de temps après la transformation. Autrement dit, le nombre d'interprétations du programme P' doit être plus grand que le nombre d'interprétations du programme P. Alors, nous pouvons supposer que la nouvelle structure obtenue n'est plus *AF-présentable*, et qu'elle possède aussi la propriété d'indépendance.

Un exemple de *structure automatique* permettant d'expliquer cette situation peut être les structures virtuellement abéliennes, c'est-à-dire les structures dont un sous-ensemble est un groupe abélien. Comme le montre Hodges, Wilfrid *et al.* [HW+93] dans les annexes de son guide sur la théorie des modèles, la théorie du premier ordre d'un groupe abélien est décidable. Un groupe abélien peut posséder la propriété d'indépendance, ce qui rend cette théorie *instable*, c'est-à-dire qu'il est possible d'utiliser cette théorie pour encoder l'*ensemble ordonné des entiers naturels* (*the ordered set of natural numbers*). Dans ce cas, si une interprétation d'un groupe fini abélien est réalisée dans une théorie du premier ordre, cette interprétation construit une suite de recouvrement.

À notre connaissance, aucun travail publié au moment de la rédaction de ce manuscrit ne propose de liens entre ces aspects de l'interprétation et les prédicats opaques. Mieux maîtriser ces aspects permettrait d'améliorer la détection des prédicats opaques. Ainsi, de cette conjecture, nous proposons dans le reste de ce chapitre une méthode pour mesurer la croissance de la suite de recouvrement d'un prédicat opaque. Nous proposons par ce résultat un nouveau distingueur pour détecter les prédicats opaques.

5.3.2 Mise en place d'une mesure de détection

Nous proposons de mettre en place la mesure suivante lors de l'analyse d'un programme par un moteur *DSE* : détecter la présence d'une structure indépendante lors de la résolution d'une requête. Nous supposons que le temps des appels de résolution est proportionnel à la vitesse de recouvrement. Pour mettre en place cette mesure, nous proposons d'utiliser les résultats des travaux de Cameron, décrit dans l'étude de Macpherson [Mac11], pour déterminer différents

taux de croissance de recouvrement de structures construit à partir de solveur (*SMT*, Théorie, SAT). Ces taux peuvent être modélisés par une fonction exponentielle. La mesure du taux de croissance d'une fonction exponentielle peut s'effectuer de deux façons différentes, mais équivalentes : par une croissance effective [28] ou par une croissance intrinsèque [29]. Ces deux mesures sont reliées par la formule suivante : $1 + \mu = e^r$, où μ représente le taux de croissance effectif et r le taux de croissance intrinsèque. Les résultats de CAMERON concernent le taux de croissance intrinsèque. En pratique, seul le taux de croissance effectif est mesurable, car nous n'avons pas d'accès facile à la taille des éléments recouverts. Ainsi, pour détecter si une structure est indépendante, dans cette modélisation, nous devons regarder si le taux de croissance effectif des appels dépasse la valeur de $ln(2) = 0{,}69$ (valeur déduite des travaux de CAMERON décrits dans [Mac11]).

L'algorithme 9 présente une manière de réaliser cette mesure. Nous intégrons cet algorithme directement dans le moteur *DSE*, et le solveur *SMT*. Lorsque le seuil de détection est dépassé, la requête *SMT* en cours d'analyse est renvoyée vers l'utilisateur du moteur *DSE*, avec une indication qu'elle possède un prédicat opaque. Nous laissons le choix des actions futures à l'utilisateur pour analyser ou modifier cette requête. Nous présentons dans l'exemple 19 un cas pratique simplifié de mesure effective.

Algorithme 9 : Détection des Prédicats opaques

Input : Time t_1, Time t_2, Time t_3, Time t_4, Int $taux$
Output : Bool $test$

```
1 test ⟵ false
2 val ⟵ Rate(t1, t2, t3, t4)
3 if val > taux then
4     test ⟵ true
5 return test
```

Exemple 19:
Pour calculer le taux de croissance effectif d'une résolution d'une requête *SMT*, réalisée par un solveur *SMT Lazy*, nous pouvons mesurer, par exemple, l'évolution des temps de chaque appel au solveur SAT. Ce temps est mesuré à partir de l'horloge du système, c'est-à-dire en réalisant la différence de deux horodatages du système. Une valeur du taux peut être obtenue à partir de 4 clauses mesurées. Soit t_1, t_2 les temps mesurés respectivement avant et après une clause, et t_3, t_4 les temps mesurés respectivement avant et après pour la clause suivante. La

28. La croissance effective mesure l'évolution dans le temps.
29. La croissance intrinsèque mesure l'évolution d'un ensemble d'éléments , ou d'une population.

croissance effective ω_1 est calculée avec la formule suivante :

$$\omega_1 = \frac{(t_4 - t_3) - (t_2 - t_1)}{(t_2 - t_1)}$$

Avec les valeurs suivantes :$t_1 = 0$, $t_2 = 2$, $t_3 = 3$ et $t_4 = 6$, et la croissance effective calculée est :$\omega_1 = \frac{(6-3)-(2-0)}{(2-0)} = 0{,}5$
Pour calculer le taux de croissance effectif τ, il faut mesurer une autre croissance effective ω_2 avec deux autres clauses, puis effectuer le calcul suivant :

$$\tau = \frac{(\omega_2 - \omega_1)}{\omega_1}$$

Par exemple, si $\omega_1 = 0{,}5$ et $\omega_2 = 0{,}7$, alors $\tau = 0{,}4$.

5.3.3 Intégration dans KLEE

Nous proposons d'intégrer l'algorithme 9 dans le moteur d'exécution symbolique et concrète KLEE, ainsi que dans certains solveurs *SMT* compatibles avec lui (Z3 et Boolector). Ces outils sont développés en C ou C++11, donc notre algorithme est développé dans les mêmes langages. L'algorithme 9 est intégré à trois endroits différents dans ce moteur d'exécution dynamique symbolique. Le premier est dans KLEE et mesure le temps de résolution pour chaque requête *SMT*. Les deux autres mesures se font dans le solveur *SMT*, lors des résolutions effectuées avec le solveurs de théorie ou le solveur SAT. La figure 5.1 présente les trois endroits où nous effectuons nos mesures de temps dans le cas de l'utilisation avec le solveur Z3, vu comme un solveur *SMT Lazy*. L'idée principale est de mesurer temporellement les prolongements des équivalences élémentaires de la requête, c'est-à-dire de mesurer le temps réalisé par le solveur *SMT* pour créer de nouveaux littéraux qui permettent l'accélération de la résolution.

5.3.4 Exemple détaillé

Nous reprenons l'exemple présenté dans la section 3.4, et nous présentons quelques mesures réalisées avec KLEE et ses différents solveurs. Nous rappelons le code source C dans le listing 5.1.

```
1 void foo(int x){
2         if(((x*(x+1)%2) != 0 )
3                 assert(false);
4         }
```

Listing 5.1 – Exemple de prédicat opaque dans un programme C

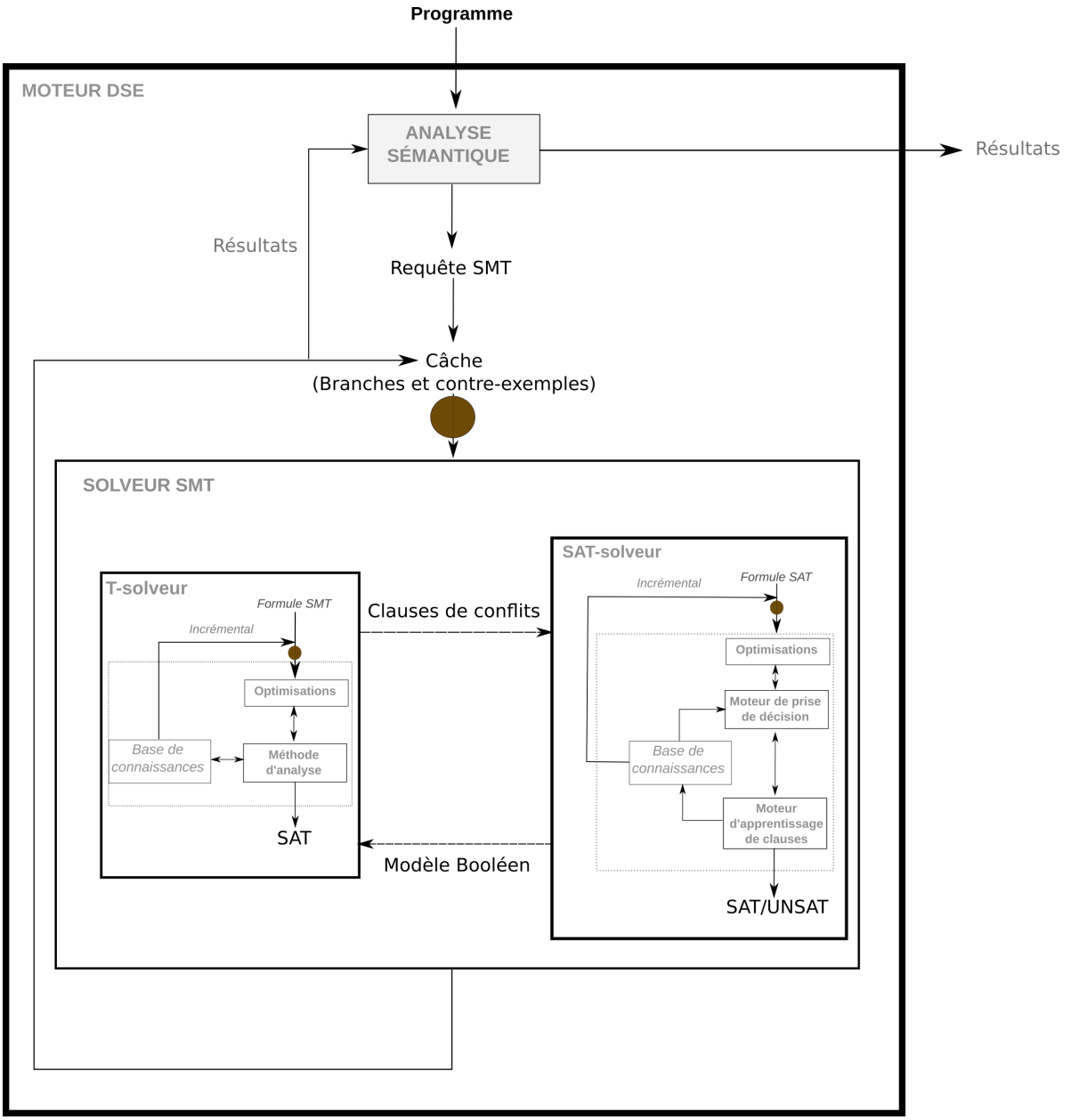


FIGURE 5.1 – Mesures temporelles aux interfaces (points marron) : cas pratique avec le solveur Z3

Mesures	KLEE	Solveur théorie	Solveur SAT
Z3	0	0,71	0,30
Boolector	0	0	0,40

TABLE 5.1 – Taux de croissance de recouvrement maximal

Les mesures obtenues sont détaillées dans le tableau 5.1. Nous observons dans ce tableau, qu'aucune *justification* n'est retournée par aucun solveur *SMT* pour cette requête. Le prédicat opaque est bien détecté avec le solveur Z3. Nous observons de plus que Boolector ne fait pas appel à son solveur de théorie, et qu'il ne rencontre pas de difficultés pour résoudre cette requête. Pour mieux comprendre cette situation, nous avons regardé plus en détail la situation rencontrée par Boolector. Pour rappel, Boolector utilise un langage interne, nommé BTOR, qui permet de traduire les expressions *SMT* en circuit et de simplifier ce circuit sans augmentation de la mémoire utilisée (*cf.* sous-section 3.3.2). Lorsque nous demandons à Boolector de nous retourner le circuit *AIG* de la requête *SMT* contenant le prédicat, l'expression est simplifiée en un circuit constant faux. Nous comprenons que Boolector, en simplifiant les expressions au *niveau mots*, capture sans difficulté le comportement constant de ce prédicat opaque. Par cet exemple, nous venons de voir que nous pouvons détecter un prédicat opaque, et l'effet opaque d'un prédicat peut être annihiler si son comportement constant est détectable au niveau mots. Dans ce cas, le prédicat n'est plus opaque et notre méthode de détection ne fonctionne pas.

5.4 Évaluation et Expériences

Nous évaluons l'efficacité de notre proposition pour identifier automatiquement différents prédicats opaques. Au moment de la rédaction de ce manuscrit, nous n'avons pas finalisé une intégration automatique de notre détecteur dans KLEE et les différents solveurs, ce qui ne nous permet pas de proposer un outil clé en main. Cette situation regrettable ne nous permet pas de réaliser un nombre important de tests. Nous essayerons dans un avenir proche de diffuser cet outil sur des plateformes de logiciels libres.

5.4.1 Évaluation

Les expériences ci-dessous visent à répondre aux questions de recherches suivantes :

QR1 : quel est le taux de succès de notre outil sur du code protégé ?

QR2 : quel est le taux de fausse alarme sur du code non protégé ?

```
void Predicate(uint64_t input_A, uint64_t *output_C){
        if((input_A & 0xffffffff80000000)==0){
                uint64_t place_holder1 = input_A & 0x7fffffff;
                uint64_t place_holder2 = (((place_holder1 >> 0x1F) - ((
    place_holder1 * 0x66666667) >> 2)) << 2) + ((place_holder1 >> 0x1F) - (
    place_holder1 * 0x66666667));
                *output_C = ((place_holder2 * 2) - place_holder1) + ((place_holder2
     * 2) - place_holder1);
        }
}
```

Listing 5.2 – Exemple de fonction insérée dans un programme par *Tigress* pour fabriquer un prédicat opaque (représentation équivalente en code C à partir d'un binaire)

5.4.2 Configuration et Jeux de données

Configuration Les expériences ont été réalisées sur une machine basée sur le CPU Intel® Core™ i7-5600U @ 2.60 GHz, avec 16 Go de *RAM*, et fonctionnant avec le système d'exploitation Ubuntu 16.04 *LTS*.

Jeux de données Pour évaluer le taux de succès (**QR1**), nous utilisons les programmes étalons (*benchmarks*) proposés par Banescu, et plus particulièrement les programmes dont le code source est connu, afin de pouvoir tester la qualité de notre détection. Nous obtenons un total de 99 fichiers, contenant des programmes de petites tailles, et représentatif de ce que l'on peut trouver dans des programmes usuels. Chaque fichier contient trois prédicats opaques, ce qui permet d'obtenir 297 cas différents de prédicats opaques. Pour information, ces prédicats opaques ont été générés à l'aide de l'outil *Tigress* [30] [Col+19]. Le listing 5.2 est un exemple en langage C, que nous avons pu extraire d'un binaire, d'une fonction insérée dans un programme par *Tigress* pour créer un prédicat opaque. Ce prédicat consiste à comparer la valeur des deux variables d'entrées de la fonction `Predicate` (une variable et un pointeur), après avoir exécuté cette fonction.

Pour évaluer le taux de faux positifs (**QR2**), nous utilisons la dernière version (8.31) des *utilitaires du noyau GNU* (*GNU core utilities*) [31] au moment de la rédaction de ce manuscrit. Ces utilitaires sont couramment utilisés dans de nombreux programmes, et sont aussi représentatifs des programmes utilisables. Pour être interprétables avec `KLEE`, nous compilons ces utilitaires avec `LLVM`, en activant l'option d'*affectation statique unique* (*single static assignment* (*SSA*)).

30. `http://tigress.cs.arizona.edu/`

31. `https://www.gnu.org/software/coreutils/`

5.4.3 Résultats et observations

Les résultats du tableau 5.2 nous permettent de vérifier la faisabilité de notre méthode de détection sur nos jeux de données. Avec cette méthode, nous détectons des prédicats dans les deux jeux de données. Quelques précisions sont nécessaires à la compréhension de ces résultats.

Mesures	KLEE (Z3)	Solveur théorie (Z3)	Solveur SAT (Z3)	KLEE (Boolector)	Solveur théorie (Boolector)	Solveur SAT (Boolector)
Banescu benchmark's	127	268	35	10	99	5
GNU core utilities	1	9	2	3	1	5

TABLE 5.2 – Quantité de prédicats opaques détectés

Pour le premier jeu de données, chaque prédicat opaque est présent dans une seule condition de branchement, ce qui nous permet de garantir que chaque requête contient au plus un seul prédicat opaque. Autrement dit, lorsque notre détection est positive, sans ambiguïté, nous venons de trouver un prédicat opaque au « bon endroit » dans le programme.

Pour le second jeu de données, si des prédicats opaques existent, nous ne pouvons ni garantir le fait qu'ils soient présents dans le programme dans une condition de branchement, ni que chaque requête en contient seulement un. Dans ce cas, la détection nous indique seulement l'existence d'au moins un prédicat opaque dans une zone spécifique du programme.

Enfin, comme nous l'avons remarqué dans notre exemple détaillé (*cf.* sous-section 5.3.4), la simplification des requêtes effectuée par `Boolector` au *niveau mots* permet dans certains cas, de supprimer l'opacité des prédicats, et rend par la même occasion inefficace notre méthode de détection. Nous observons dans le tableau 5.2 que la quantité de prédicats opaques détectés globalement est plus importante pour le solveur `Z3` que pour le solveur `Boolector`. Cette simplification permet de distinguer clairement l'existence de prédicats dont le comportement constant est au *niveau mots* ou au *niveau bits*. Autrement dit, suivant les outils utilisés, nous ne détectons pas exactement les mêmes prédicats opaques : avec `KLEE`, nous pouvons détecter les prédicats opaques dont le comportement constant est décrit au *niveau phrase*, *mots* et *bits*, avec `Z3` et son solveur SAT, nous détectons les prédicats opaques dont le comportement constant est décrit au *niveau mots* et *bits*, et avec `Boolector` et son solveur SAT, nous pouvons faire la distinction entre certains prédicats opaques dont le comportement constant est décrit au *niveau bits* et ceux dont le comportement constant est décrit au *niveau mots* et au *niveau bits*. Une instrumentalisation plus importantes des solveurs *SMT* pourrait aider à isoler et classer les prédicats opaques en fonction de leur niveau d'opacité .

5.5 Discussions

5.5.1 Méthodologie

Nous discutons de l'existence de biais dans notre évaluation expérimentale.

Métrique Le calcul de l'évolution du taux de recouvrement des présentations nécessite un minimum de quatre présentations. Nous ne pouvons pas dire précisément quelle présentation est une structure indépendante.

Ensemble de prédicats opaques Nous avons utilisé un ensemble de programmes contenant des prédicats opaques. La littérature actuelle ne fournit pas de méthodes pour classer la force des prédicats opaques. L'évaluation de notre technique permet de confirmer que celle-ci fonctionne contre des prédicats opaques connus, considérés comme des représentants de situations réelles. Mais qu'en serait-il pour d'autres prédicats ?

Qualité de l'analyse statique La qualité de l'analyse statique dépend de la précision de résolution du solveur *SMT*. La présence d'un prédicat opaque dans la requête analysée va dépendre de cette analyse. Si la représentation est imparfaite, et qu'une relation n'est pas capturée par le langage, cela peut créer les mêmes effets qu'un prédicat opaque. Dans ce cas, notre méthode peut détecter la présence de cette non-relation, mais elle ne rentre pas dans la définition d'un prédicat opaque. Cette situation apparaît seulement pour au moins un analyseur. Ce qui augmente le taux de faux négatif pour notre méthode.

Interfaces Pour mettre en place notre méthode, il faut que les différents composants de la résolution d'une requête puissent être accessibles. Si ces composants sont correctement séparés en différentes bibliothèques, un développeur expérimenté mettrait peu de temps pour intégrer notre méthode à l'outillage disponible. Malheureusement, au niveau des solveurs *SMT*, l'interface avec les solveurs SAT n'est pas toujours des plus simples, car tous les projets de solveurs *SMT* ne sont pas strictement modulaires.

5.5.2 Limites de cette méthode et pistes futures

Représentation et temps de détection En fonction de la stratégie d'analyse d'une présentation automatique, le temps de détection d'une structure indépendante peut être long. Pour un

prédicat opaque fixé, il peut exister différentes représentations contenant au moins une structure indépendante. Nous ne savons pas s'il est possible de trouver une méthode permettant de borner le nombre d'étapes permettant de trouver cette structure. Mais, même sans avoir cette méthode, nous espérons qu'au moins l'une des représentations du prédicat permet une détection plus tardive de la structure indépendante.

Représentation symbolique La *rétro-ingénierie* d'un programme s'effectue habituellement depuis l'analyse d'un binaire. Ainsi, les prédicats opaques peuvent agir aussi sur la mise en place de la représentation symbolique, afin de rendre cette dernière imparfaite. L'utilisation de `KLEE` suppose que nous obtenons le cas parfait de la représentation symbolique d'un programme. Nous ne savons pas dans quelle mesure notre méthode reste valable pour une analyse symbolique et concrète de binaires. Nous pouvons supposer que l'utilisation de notre méthode telle quelle n'est pas suffisante en général. L'analyse de binaires est réalisée à partir de traces d'exécutions. Nous pouvons facilement comprendre que ces traces peuvent ne pas être suffisantes pour capturer le comportement entier d'un programme, ce qui peut créer une *décohérence*. Cette *décohérence* pourrait générer des structures possédant la propriété d'indépendance, ce qui augmenterait notre taux de faux positifs. Une réflexion sur la quantité minimale d'informations à récupérer symboliquement pourrait aider à mieux comprendre cette situation.

Nous avons testé notre méthode sur un petit nombre d'échantillons. Une piste future serait de pouvoir tester sur un nombre plus important, avec une variété de programmes plus importante, comme des *logiciels malveillants* (*malwares*).

5.6 Conclusion du chapitre

L'utilisation des prédicats opaques comme méthode d'obfuscation permet d'augmenter le temps d'interprétation réalisé lors de l'analyse d'un programme. Les attaques récentes basées sur les outils d'analyse automatique ont pu prouver leur efficacité face à cette technique (mais aussi ses faiblesses). Nous avons pu montrer l'importance d'étudier les mécanismes de fonctionnement des outils d'analyse basés sur l'utilisation d'une théorie du premier ordre. Sur cette connaissance, nous proposons une nouvelle méthode pour détecter des prédicats opaques, vus comme des structures indépendantes. Nous proposons une conjecture sur la structure intrinsèque des prédicats opaques. À partir de cette conjecture, nous intégrons dans `KLEE` et deux solveurs une mesure sur la résolution des requêtes. Nous testons notre méthode sur un ensemble de programmes ne contenant et ne contenant pas de prédicat opaque. À partir de ce travail, nous

espérons pouvoir proposer un outil permettant de détecter les prédicats opaques connus, mais aussi inconnus.

Chapitre 6

Notre proposition pour réduire l'opacité d'un prédicat opaque

Sommaire

6.1 Introduction 130
6.2 Prédicat et représentations 136
6.3 Redéfinition d'une expression *SMT* 138
6.4 Évaluations et expériences 141
6.5 Travaux existants 143
6.6 Conclusion du chapitre 144

Résumé Nous proposons une méthode pour redéfinir un prédicat opaque contenu dans une requête *SMT*, dans le but de réduire le temps d'analyse nécessaire à un moteur *DSE*. Nous fixons un contexte d'interprétation basé sur la théorie QF-ABV (ou QF-AUFBV). Nous conjecturons que l'interprétation dans cette théorie d'un prédicat opaque peut générer une relation de définition *instable*. Notre objectif est de stabiliser cette relation de définition en ordonnant l'analyse du prédicat. En pratique, cela se traduit par la redéfinition de certaines opérations, et de sous-requêtes *SMT*. Autrement dit, nous adaptons l'équilibre entre la représentation des structures du prédicat et l'efficacité de la résolution proposée par un moteur d'*exécution symbolique dynamique* à l'analyse d'un prédicat opaque. Nous nous intéressons dans un premier temps à l'interprétation d'un prédicat et de ses solutions dans une théorie homogène, puis nous présentons une méthode de redéfinition modifiant cette interprétation. Enfin, nous testons notre méthode sur des programmes et observons que nous arrivons à réduire le temps d'analyse de ces programmes. Ce chapitre contient les principaux résultats des travaux réalisés en collaboration avec Ioana CRISTESCU (Inria), Olivier DECOURBE (Inria), Sébastien JOSSE (Dga), et Caroline FONTAINE (CNRS).

6.1 Introduction

Un attaquant défini dans le modèle *des attaques de l'homme à la fin* (*MATE*) utilise un *moteur d'exécution symbolique dynamique* (*DSE*) pour comprendre le comportement d'un programme exécutable obfusqué à l'aide de prédicats opaques. Les prédicats opaques sont des objets sémantiques utilisés dans un programme afin de ralentir la compréhension de ce dernier par un adversaire, mais pas ou peu son exécution. Chaque prédicat se comporte toujours de la même manière pour toutes les exécutions, c'est-à-dire qu'un prédicat opaque est toujours *vrai* ou toujours *faux*. Plus précisément, l'utilisation d'un prédicat opaque en tant que technique d'obfuscation cache le comportement d'une condition de branchement définie dans le *graphe de flot de contrôle* (*CFG*) du programme. Alors, lorsqu'un *moteur d'exécution symbolique dynamique* rencontre un prédicat opaque, le temps qu'il lui faut pour déduire le comportement constant de ce prédicat est « anormalement long ». Parfois, la requête contenant le prédicat opaque ne peut pas être résolue, et seule une concrétisation du prédicat permet d'éviter son opacité. Nous conjecturons qu'un prédicat opaque peut générer une relation de définition *instable*, c'est-à-dire une interprétation *instable*. Or, les précédentes méthodes vues dans la section 2.3.2 pour réduire les effets des prédicats opaques s'intéressent à la représentation des prédicats et non à son interprétation. Nous souhaitons explorer cette nouvelle piste, dans le but de savoir s'il existe au moins une représentation d'un prédicat opaque moins *instable* qui atténue son opacité.

6.1.1 Problème

Peut-on modifier la définition d'un prédicat opaque représenté dans une requête SMT telle que réalisée par un moteur DSE, pour obtenir un temps de décision plus rapide ? Une analyse de programme réalisée à l'aide d'un moteur *DSE* représente un programme par un ensemble de contraintes, définies dans un langage décidable. Résoudre ces contraintes permet de connaître le comportement du programme. Lorsqu'un ensemble de ces contraintes ne peuvent être résolues facilement par déduction, cet ensemble de contraintes est traduit dans une requête définie dans un langage du premier ordre, pour être résolue à l'aide d'un solveur *SMT*. En pratique, ce langage du premier ordre est basé sur une théorie décidable, complète et homogène (par exemple : QF-ABV ou QF-AUFBV). Ces théories permettent de capturer les structures de données manipulées dans les registres ou les mémoires, en représentant ces données sous forme de vecteurs booléens de tailles finies. Pour comprendre ce que le terme « anormalement long » veut dire dans le cas d'une analyse automatique d'un prédicat opaque, nous avons étudié le comportement de différents moteurs *DSE*. De cette étude, nous pouvons proposer deux scénarios comportementaux des

interactions entre le moteur *DSE* et le solveur *SMT* lors de l'analyse d'un prédicat opaque. Le premier scénario est le suivant : le temps d'analyse d'une requête *SMT* est long et demande une quantité de mémoire importante et finie. Nous supposons de cette observation que le nombre de clauses utilisées par le solveur *SMT* est important. Nous décrivons maintenant le second scénario. Le moteur *DSE* construit une requête *SMT*. Cette requête n'est pas résoluble par le solveur, mais ce dernier retourne la *justification* (appelé aussi le *cœur* (*core*)) de pourquoi cette requête n'est pas résoluble. La conjonction de la négation de cette *justification* avec la requête *SMT* permet de créer une nouvelle requête. La taille de la nouvelle requête est plus grande que l'ancienne, c'est-à-dire que le nombre de relations définies dans la nouvelle requête est plus grand que le nombre de relations définies dans l'ancienne requête. La nouvelle requête est soumise au solveur *SMT*, et la situation précédente se répète jusqu'à ce que la requête obtenue devienne résoluble. La taille de la requête *SMT* croît de manière exponentielle, ce qui allonge de la même manière le temps de résolution. Ajoutons enfin quelques détails dans ce scénario. Nous supposons que le moteur *DSE* ne limite pas le temps d'utilisation du solveur *SMT*, et qu'il ne prend pas la décision de concrétiser les contraintes définies dans la requête. Nous supposons aussi que le nombre de requêtes obtenu est important, et qu'il n'existe pas de moyen de prédire ce nombre [1]. De ces scénarios, nous pouvons comprendre facilement que l'interprétation par un moteur *DSE* des relations existant dans le prédicat influence la définition de ces relations exprimées dans la requête *SMT*, autrement dit, l'interprétation réalisée par un moteur *DSE* influence le temps de l'analyse. Alors, trouver une méthode pour qu'un moteur *DSE* puisse plus facilement analyser un prédicat opaque correspond à une instance du problème *SMT*, dans laquelle la fonction d'interprétation est fixée. Pour aider cette fonction d'interprétation, nous pouvons travailler sur la définition des relations constituant le prédicat, ce qui nous amène à notre question énoncée en début de paragraphe.

6.1.2 Objectifs et contributions

Notre principal objectif est de proposer une méthode pour réduire le temps d'analyse de programmes protégés par des prédicats opaques. Cette méthode doit être générique, sans erreurs, et facilement intégrable dans un moteur *DSE* utilisant un solveur *SMT*. Notre second objectif est de proposer une analyse plus précise des effets des prédicats opaques dans un moteur *DSE*. Cette analyse apporte des éléments de réponses sur l'origine des effets observés. Nous résumons les contributions de ce chapitre de la manière suivante :

1. À notre connaissance, pour une théorie du premier ordre, il n'existe pas de méthode permettant de connaître à l'avance le nombre de requête permettant de reconstruire un modèle.

- nous proposons une vision plus formelle des effets des prédicats opaques,
- nous proposons un système simple de redéfinition des requêtes *SMT* contenant un prédicat opaque,
- nous testons notre système sur un ensemble de programmes contenant différents prédicats opaques.

Le reste de ce chapitre est structuré de la manière suivante. La section 6.2 présente les aspects formels de la stabilité de l'interprétation d'une structure dans une théorie. Nous présentons section 6.3 notre méthode de redéfinitions. Nous réalisons ensuite dans la section 6.4 quelques tests de notre méthode. Nous terminons ce chapitre en présentant l'ensemble des travaux relatifs à la redéfinition des prédicats opaques dans la section 6.5.

6.1.3 Exemple type

Nous proposons un exemple type pour illustrer le fonctionnement de notre méthode de redéfinition. Le listing 6.1 présente un programme C obfusqué, contenant un faux test de mot de passe. Ce programme contient deux prédicats invariants, renforcés à l'aide d'une fonction de hachage non cryptographique `APhash` [Par13] (listing 6.2). Pour simplifier l'analyse, la taille des entrées est fixée à 32-bits. Le programme est ensuite compilé à l'aide de *clang*, avec optimisations (c'est-à-dire l'option `-O3`) et l'option l'option d'*affectation statique unique* (*SSA*). Le programme est ensuite analysé avec un moteur d'exécution symbolique dynamique (`KLEE` ou `angr`). Ce dernier fait appel à un solveur *SMT* pour résoudre des conditions de branchement, à l'aide de requête définie dans la théorie QF-ABV (ou QF-AUFBV). Un exemple de requête *SMT* généré par `angr` est donné dans le listing 6.3 [2].

Nous conjecturons que nous utilisons la méthode de détection de prédicat opaque du chapitre précédent. Dans cet exemple le prédicat opaque est défini à la ligne 25 du listing 6.1. Nous proposons maintenant de voir comment nous procédons pour récrire une requête *SMT*. Notre module de redéfinition est développé en Python. Nous analysons la représentation symbolique de la requête *SMT*. Nous prenons l'expression définie ligne 7 du listing 6.1 : `var = (var * var + var)% 2`. Cette expression est une définition de la variable `var`. Nous regardons indépendamment du reste du programme, les valeurs que peut prendre cette variable à l'aide du moteur *DSE*. L'opérateur modulo nous indique qu'il existe seulement deux valeurs. Nous concrétisons ces deux valeurs dans deux variables différentes à l'aide du langage Python, et nous calculons la valeur de var à partir de chacune de ces valeurs. Nous observons que nous obtenons la même valeur numérique

2. Nous ne mettons pas le fichier original, car il fait plus de 6 Mo.

```c
#include <stdio.h>

int Func1(char* in1){
  unsigned int hash = 0xAAAAAAAA, i = 0, var = 0;
  for (i = 0; i < 32; ++i){
        var = *in1;
        var = (var*var+var)%2);
        hash ^= ((i&1)==0)?((hash<<7) ^ (var) * (hash>>3)):(~((hash<<11) + ((var) ^ (hash>>5)
    )));
  }
  return hash;
  }

int Func2(char* in1){
  unsigned int hash = 0xAAAAAAAA, i = 0, var = 0;
  for (i = 0; i < 32; ++i){
        var = *in1;
        var = (var*var+25)%4));
        hash ^= ((i&1)==0)?((hash<<7) ^ (var) * (hash>>3)):(~((hash<<11) + ((var) ^ (hash>>5)
    )));
  }
  return hash;
  }

int main(int argc, char *argv[]){
   if(argc == 3){
        if(!(Func1(argv[1]) != Func2(argv[2])))
        {
           printf("SUCCESS\n");
           return 1;
        }
   }
   printf("FAIL\n");
   return 0;
}
```

Listing 6.1 – Exemple type

```
1  unsigned int APHash(const char* str, unsigned int length)
2  {
3     unsigned int hash = 0xAAAAAAAA;
4     unsigned int i    = 0;
5
6     for (i = 0; i < length; ++str, ++i)
7     {
8        hash ^= ((i & 1) == 0) ? (  (hash <<  7) ^ (*str) * (hash >> 3)) :
9                                 (~((hash << 11) + ((*str) ^ (hash >> 5))));
10    }
11
12    return hash;
13 }
```

Listing 6.2 – Fonction de hachage `APhash` [Par13] « `Copyright 2020 Arash Partow` » `General Hash Function License` selon `MIT License` (`https://www.partow.net/programming/hashfunctions/#GeneralHashFunctionLicense`)

0. Nous pouvons remplacer la définition symbolique de var par sa définition numérique 0. Puis, nous propageons cette nouvelle définition dans le reste de la requête pour la simplifier. Nous observons alors qu'il n'existe aucun degré de liberté pour la fonction `Func1`, nous pouvons aussi la concrétiser avec le langage Python. Nous remplaçons la définition de la fonction `Func1` par la valeur obtenue.

La deuxième partie consiste ensuite à remplacer les opérations de *translations* (*shift*), par des opérations d'additions, pour redéfinir la taille des variables. Par exemple, l'opération définie ligne 18 : `hash « 7` permet de redéfinir la variable `hash`, par l'expression symbolique $hash_1 * 2^{25} + hash_2$, avec $hash_1$ une variable définie sur 7-bits et $hash_2$ une variable définie sur 25-bits. Ce qui permet de remplacer l'opération symbolique équivalente à `hash « 7` par seulement $hash_2$. Dans certains cas, les multiplications peuvent être aussi remplacées par une addition, mais ce n'est pas utile ici.

La troisième partie consiste à résoudre par *tranches* (*slices*) la requête *SMT*, c'est-à-dire *vérifier* (*to check*) la satisfiabilité de chaque tour de boucle de la fonction `Func2` une par une. Pour chaque requête satisfaite, nous demandons un modèle au solveur, pour mettre à jour la valeur de la variable $hash$. Si l'ensemble des sous-requêtes sont satisfaites, alors la requête *SMT* est satisfaite. Le temps de résolution de l'ensemble de ces sous-requêtes est beaucoup plus faible que le temps de résolution de la requête *SMT*.

```
(set-info :status unknown)
(set-logic QF_AUFBV)
(declare-fun symbo2_9_32 () (_ BitVec 32))
(declare-fun symbo1_8_32 () (_ BitVec 32))
(assert
 (let ((?x5372 (bvadd (_ bv1 1) (bvmul ((_ extract 24 24) symbo2_9_32) ((_ extract 24 24) symbo2_9_32)))))
(let ((?x11627 (bvadd (_ bv1 2) (bvmul ((_ extract 25 24) symbo2_9_32) ((_ extract 25 24) symbo2_9_32)))))
(let ((?x6968 (bvmul (_ bv85 8) (concat (_ bv0 6) ?x11627))))
(let ((?x10855 (bvnot ((_ extract 1 1) ?x6968))))
(let ((?x6377 ((_ extract 2 2) ?x6968)))
(let ((?x6599 (concat ?x6377 ?x10855)))
(let ((?x8357 (concat ?x6599 ?x5372)))
(let ((?x14085 (bvadd (_ bv1 2) (bvmul ((_ extract 17 16) symbo2_9_32) ((_ extract 17 16) symbo2_9_32)))))
(let ((?x12987 (concat (_ bv0 12) ?x14085)))
(let ((?x9916 ((_ extract 5 5) ?x6968)))
(let ((?x10454 (bvnot ?x9916)))
(let ((?x14828 ((_ extract 6 6) ?x6968)))
(let ((?x8299 ((_ extract 7 7) ?x6968)))
(let ((?x16352 (bvnot ?x8299)))
(let ((?x7773 (bvmul (_ bv357913941 32) (concat (_ bv0 30) ?x11627))))
(let ((?x13075 ((_ extract 8 8) ?x7773)))
(let ((?x15131 (bvnot ?x13075)))
(let ((?x14977 ((_ extract 9 9) ?x7773)))
[...]
```

Listing 6.3 – Début de la requête *SMT* fabriquée par `angr`

6.2 Prédicat et représentations

L'analyse statique d'un programme sans métadonnées est en général indécidable. Dans cette section, nous présentons le contexte formel qui nous permettra de redéfinir les expressions des requêtes. Nous rappelons dans un premier temps quelques résultats connus sur les prédicats et les idéaux. Puis, nous présentons les caractéristiques d'une interprétation *instable*.

6.2.1 Prédicat et solutions

Lorsqu'un prédicat est défini par une requête exprimée dans une théorie décidable, le solveur de cette théorie peut retourner un modèle de cette requête si celle-ci est satisfiable. Dans le cas d'une *théorie homogène*, nous avons vu dans le précédent chapitre que ce modèle correspond à la construction d'une *équivalence élémentaire* entre la requête et la théorie. Or, un *idéal* (*ideal*) en théorie des ensembles, représente une collection de plusieurs ensembles partiellement ordonnés considérés comme petits en cardinalité. De plus, comme un modèle peut être défini à partir de son ensemble de définition, cet ensemble peut être recomposé à partir d'idéaux. Ainsi, nous pouvons supposer qu'un modèle d'une requête contenant un prédicat correspond à une composition d'*idéaux* de l'ensemble de définitions du prédicat. Autrement dit, si les prédicats opaques sont vus comme une fonction booléenne définie depuis l'*ensemble partiellement ordonné* (*poset*) des états d'un programme vers l'ensemble $\{0,1\}$, alors les solutions de ce prédicat sont composées à partir d'un sous-ensemble des *idéaux* de ce *poset*. Dans ce cas, nous pouvons proposer une nouvelle définition des prédicats opaques.

Définition 6.1 *Soit un ensemble partiellement ordonné $(L,\leq)$, un* prédicat opaque *est une structure automatique sans symbole de fonction, représentant un sous-ensemble L' de L. L'interprétation d'un prédicat opaque dans une théorie décidable est réalisée à l'aide d'une collection d'idéaux de L'.*

Cette définition nous permet de supposer qu'il existe une équivalence entre l'ensemble des solutions d'un prédicat et la taille du modèle calculable de ce prédicat. Nous n'avons pas trouvé de référence pouvant infirmer ou confirmer cette hypothèse. Cependant, nous avons compris que dans le cas de l'utilisation de la théorie QF-ABV (ou QF-AUFBV), plusieurs interprétations équivalentes sont possibles.

6.2.1.1 Concept d'équivalence

Nous regardons maintenant plus en détail le concept d'équivalence. Une expression logique φ, définie dans une théorie $\mathcal{T}$, est dite *satisfiable* s'il existe un modèle $\mathcal{M}$ qui est un modèle de φ. Deux expressions ϕ et φ sont dite *équi-satisfiables* (ou modèle équivalent) si ces deux formules possèdent un même modèle $\mathcal{M}$, c'est-à-dire si φ est satisfiable pour $\mathcal{M}$ quand ϕ est satisfiable pour $\mathcal{M}$, et réciproquement. Dans ce cas, $\mathcal{M}$ est une équivalence entre l'expression φ et la théorie $\mathcal{T}$, et M est aussi une équivalence entre l'expression ϕ et la théorie $\mathcal{T}$.

D'autre part, en algèbre, une relation d'équivalence peut être définie dans un ensemble, si elle possède les propriétés suivantes : *réflexivité*, *symétrie*, et *transitivité*. Pour une relation d'équivalence, nous pouvons définir une *classe d'équivalence*, puis la notion d'équivalence (algébrique) suivante : deux éléments sont dits *équivalent* si et seulement si ces éléments appartiennent à la même *classe d'équivalence*. Alors, puisqu'un modèle permet de définir une équivalence, nous pouvons supposer qu'il existe un lien entre un modèle et une *classe d'équivalence*. Ainsi, comprendre comment un modèle est construit revient à comprendre comment parcourir les éléments d'une *classe d'équivalence*.

6.2.1.2 Structure ω-catégorique

Pour une structure $\mathfrak{A}$, son *automorphisme* est une application bijective de l'ensemble de définition de $\mathfrak{A}$, noté A, dans lui-même, qui préserve les relations et les fonctions de $\mathfrak{A}$. Plus précisément, un *automorphisme* π de $\mathfrak{A}$ peut être appliqué à un k-uplet $\bar{a}$ défini par $(a_1, a_2, \ldots, a_k)$, pour des éléments a_i de l'ensemble A, produisant comme résultat $\pi(\bar{a})$ défini par le k-uplet $(\pi(a_1), \pi(a_2), \ldots, \pi(a_k))$, pour tout entier $k > 0$. Alors, deux k-uplets $\bar{a}$ et $\bar{b}$ sont dit de la même *orbite* s'il existe un *automorphisme* qui relie $\bar{a}$ à $\bar{b}$. Une *orbite* est une *classe d'équivalence* de cette relation d'équivalence. Alors, une structure dénombrable $\mathfrak{A}$ est dite ω-*catégorique* si pour chaque $n \in \mathbb{N}$, l'ensemble des uplets A^n possèdent un nombre fini d'*orbites*.

> **Exemple 20:**
> Les théories QF-ABV ou QF-AUFBV sont des théories ω-*catégoriques*.

6.2.2 Vérité et stabilité de l'interprétation

Nous avons pu observer au travers des exemples précédents que l'interprétation des prédicats opaques dans une théorie ω-catégorique fixée peut être *instable*, c'est-à-dire que suivant le moteur d'*exécution symbolique dynamique* utilisé, et/ou suivant le solveur *SMT* choisi, la méthode

de résolution peut donner plusieurs réponses. Lors de la redéfinition des expressions *SMT*, nous devons éviter cette instabilité. Dans ce but, nous définissons maintenant les liens entre interprétation et stabilité, et nous verrons dans la section suivante comment nous pouvons renforcer ce lien pour atténuer les effets d'un prédicat opaque.

Une expression e est constituée de variables appartenant à un ensemble de variables fixées, et de relations définies dans cet ensemble. Soit une expression e avec des variables libres V, alors chaque évaluation (ou concrétisation) $val : V \rightarrow A$ définit d'une certaine manière une valeur $X = e[val]$, qui est un élément, ou un ensemble d'éléments, de A, formellement défini par induction sur la structure de e. Alors, X est dit *définissable* dans A. Plus généralement, on dira que X est *interprétable* dans une théorie $\mathcal{T}$ [3] si pour un ensemble S définissable dans une théorie $\mathcal{T}'$, il existe une relation d'équivalence E dans S. Alors, l'interprétation d'un prédicat défini dans une théorie $\mathcal{T}$ est correctement interprétée dans une théorie $\mathcal{T}'$ s'il n'existe pas un certain ordre pour cette interprétation. Sinon, cette interprétation est dite *instable* [HW+93].

6.3 Redéfinition d'une expression *SMT*

Nous conjecturons que l'interprétation d'un prédicat opaque génère une interprétation *instable*. Pour remédier à cette instabilité, nous proposons de mettre en place un ordonnancement de la résolution d'une requête *SMT*. Notre méthode est inspirée de l'action d'*étalements de bits* (*bit-blasting*) et de techniques de *découpage* (*slicing*). Nous supposons que la détection d'un prédicat opaque s'est faite par la méthode décrite dans le chapitre précédent. Par définition, une requête contenant un prédicat opaque est adaptée pour être calculable facilement, mais pas décidable facilement. Pour réduire le temps d'analyse d'une requête contenant un prédicat opaque, nous souhaitons définir le prédicat opaque autrement. En pratique, une des techniques pour supprimer l'opacité du prédicat est d'utiliser la technique d'*étalement des bits*. Bien que cette technique permette d'obtenir des temps de réponse plus raisonnables, cette technique peut se traduire par une explosion exponentielle de l'espace nécessaire pour représenter la requête. Nous proposons de décomposer la structure pour mettre en avant la structure booléenne présente dans la requête, sans modifier l'ensemble des solutions du prédicat. Nous allons voir dans le reste de cette section comment nous pouvons décomposer une requête *SMT*, puis la méthode que nous proposons.

3. Une théorie $\mathcal{T}$ est définie pour un langage L du premier ordre.

6.3.1 Décomposition d'une expression *SMT*

Lors d'une analyse statique, le comportement du programme est représenté sous forme d'expressions symboliques définies dans une théorie décidable. Une fonction d'interprétation calculable permet de donner un sens à ces expressions. L'obfuscation d'un programme a pour effet d'allonger le temps de l'interprétation. La redéfinition des expressions a pour effet de réduire le temps de l'interprétation, en simplifiant cette interprétation. Plusieurs familles de simplifications d'expressions symboliques existent. Nous nous limitons à une méthode permettant d'obtenir une représentation équivalente de l'expression *SMT* et qui rend l'interprétation plus simple. Trois niveaux de redéfinition sont possibles (phrases, mots, bits). Ces trois niveaux de redéfinition ne permettent pas en apparence d'utiliser les mêmes approches pour la redéfinition des présentations, car les difficultés rencontrées ne sont pas les mêmes. Par exemple, au niveau des bits, lorsque le nombre d'opérations arithmétiques augmente, les dépendances entre les bits augmentent aussi. Cette dépendance nécessite plus de symboles pour être représentée. Enfin, rappelons que la résolution de contraintes à l'aide d'un solveur *SMT* correspond à une solution technique du défi de calculabilité existant entre la représentation de l'information et son temps de décision, ceci implique que la simplification des expressions au niveau des bits n'est pas une solution envisageable. Pour les notions théoriques sur la redéfinition d'expression, nous utilisons les informations contenues dans la thèse de Eyrolles [Eyr17].

6.3.2 Méthode

Notre méthode consiste principalement à changer les définitions des variables, et de certaines relations, pour résoudre la requête *SMT* par un ensemble de sous-requêtes. Nous proposons d'appliquer les étapes suivantes pour améliorer l'analyse des prédicats opaques, après leur détection par la méthode proposée dans le précédent chapitre :

1. détecter une requête *SMT* contenant un prédicat opaque
2. remplacer et propager les valeurs constantes,
3. redéfinir les variables et les opérations de la requête,
4. résoudre un ensemble de sous-requêtes *SMT*,
5. continuer l'analyse.

6.3.2.1 Première étape : concrétisation des variables

Nous regardons si nous pouvons connaître la valeur de certaines variables. Si cette valeur est unique et calculable, nous remplaçons la définition symbolique de cette variable par sa valeur numérique. `KLEE` et `angr` utilisent des méthodes similaires qui s'appuient sur le solveur *SMT* pour réaliser cette étape. Nous avons fait le choix de proposer une nouvelle implémentation de cette méthode en Python pour bien maîtriser cette étape.

6.3.2.2 Deuxième étape : redéfinition des opérations

Nous redéfinissons certaines opérations pour obtenir plus de relations définies avec l'opérateur d'addition, telle que proposée dans la table 6.1. Ces règles de redéfinition augmentent le nombre de variables, mais aussi le nombre d'additions présente dans les expressions. Le but de ces règles de redéfinition est de rendre les relations plus géométriques, c'est-à-dire de réduire le nombre d'interprétations possible des opérations arithmétiques et booléennes, en prenant en compte l'ordre des opérations. Les outils d'analyses utilisés prennent en compte partiellement ce contexte, mais dans notre situation, il nous a semblé évident qu'il était nécessaire de renforcer la définition de certaines relations pour réduire le temps d'analyse.

$\mathbf{Shift-L}$	$\dfrac{x \ll y \,\vert\, T = b \cdot 2^y \quad x : T \quad y : T}{(x \ll_T y) \rightarrow (b \cdot 2^y + 0)}$
$\mathbf{Shift-R}$	$\dfrac{x \gg y \,\vert\, T = b \quad x : T \quad y : T}{(x \gg_T y) \rightarrow (0 \cdot 2^{\vert T\vert - y} + b)}$
$\mathbf{Mult32_{pair}}$	$\dfrac{y\%2 = 0}{(x \odot y) \rightarrow (x \ll_T 2^{ln(x)/ln(2)})}$
$\mathbf{Mult32_{impair}}$	$\dfrac{y\%2 = 1}{(x \odot y) \rightarrow (x \odot (y-1) + x)}$

TABLE 6.1 – Règles de redéfinition des opérations

Exemple 21:
L'expression suivante '$x \ll 3 + x \gg 2$', avec une variable x, définie sur 4 bits, peut être remplacée à l'aide des deux premières règles, par l'expression équivalente suivante '$a + c \cdot 2^3$', où la variable x est redéfinie à l'aide des variables a, b, c, respectivement de taille 2 bits, 1 bit, 1-bit, tel que $x = a \cdot 2^2 + b \cdot 2^1 + c$.

6.3.2.3 Troisième étape : redéfinition de la requête

Enfin, nous terminons en décomposant la requête *SMT* en un ensemble de sous-requêtes *SMT* plus facile à résoudre. Dits autrement, nous résolvons par parties la requête *SMT* originale afin de construire une composition de modèles. Nous utilisons la clôture de congruence pour décomposer les opérations non pas bit à bit, mais sur des ensembles de tailles finis plus petits que 32-bits. La première requête est résolue seule, et si elle est satisfiable, un modèle est demandé. Ce premier modèle est propagé dans la requête *SMT* originale, et une deuxième requête est construite de la même manière. Nous supposons que la résolution de l'ensemble de ces sous-requêtes est plus rapide à résoudre qu'une résolution directe de la requête originale. Le procédé est répété jusqu'à ce qu'il ne soit plus possible de générer de requêtes, ou que la requête devienne non satisfiable. Dans ce dernier cas, nous récupérons le cœur de la résolution, pour recommencer le processus à la première étape.

6.3.3 Implémentation

Au moment de la rédaction de ce manuscrit, nous n'avons pas pu finir le développement d'un outil automatique permettant de redéfinir les requêtes *SMT*. L'idée de cet outil, développé en Python, est d'utiliser le module AST [4] pour représenter les expressions avec Python AST. Puis, à partir du module `sympy` [5] réaliser des opérations symboliques et de simplifications d'expressions. L'outil doit être composé d'un module de prétraitement, d'un module de détection, et d'un module de redéfinition. Le module de redéfinition accepte en entrée une chaîne de caractères, ou un fichier contenant des chaînes de caractères, qui représente(nt) des formules exprimées dans une théorie homogène, au format *SMT-LIB* [BST+10], et retourne une chaîne de caractères, ou un fichier, dans le même format. La boucle principale doit simplifier les opérations arithmétiques, et redéfinir les opérations présentes dans les formules et dans la requête. L'opération de simplification arithmétique peut être appliquée avant ou après l'opération de redéfinition. Le prétraitement permet de réaliser des opérations de transformations pour « normaliser » les expressions.

6.4 Évaluations et expériences

Nous testons l'efficacité de notre méthode sur un ensemble de programmes protégés par des prédicats opaques, pour répondre aux questions de recherche suivantes :

4. AST : Arbres Syntaxiques Abstraits (*Abstract Syntactic Tree*)
5. https ://www.sympy.org

QR3 : possède-t-on un avantage temporel lorsque des règles de redéfinitions sont utilisées sur une requête *SMT* contenant un prédicat opaque ?

QR4 : en cas de mauvaise détection du prédicat opaque, est-ce que l'utilisation de ces règles allonge le temps d'analyse d'une requête *SMT* ?

Nous supposons que nous sommes dans une attaque *MATE* pour nos exemples. Nous n'avons pu tester notre méthode que sur deux exemples pour le moment, utilisant la fonction de hachage non-cryptographique `APhash`. Les moteurs *DSE* utilisés sont `KLEE` ou `angr`. Nous utilisons deux versions différentes de `KLEE` : celle avec le solveur *SMT* `Z3` et celle avec le solveur `MetaSMT` [6] qui permet l'utilisation du solveur `Boolector`. Chaque programme est compilé à l'aide de `Clang` [7] [Lat08] (ver. 6.0.0 [18]). Les expériences ont été réalisées sur une machine basée sur le CPU Intel® Core™ i7-5600U @ 2.60 GHz, avec 16 Go de *RAM*, et fonctionnant avec le système d'exploitation Ubuntu 16.04 *LTS*.

Comme nous ne possédons pas d'outil nous permettant de réaliser nos expériences automatiquement, nous avons adapté en partie les solveurs et moteurs *DSE*, pour réaliser chaque expérience. Les principales modifications sont décrites ci-après. Nous intégrons notre détecteur de prédicat opaque décrit dans le chapitre précédent dans `KLEE`, `angr`, `Z3` et `Boolector`. Pour être à l'interface entre le moteur et le solveur, nous nous plaçons dans `KLEE` au niveau de la pile de solveur pour intercepter les formules et requêtes *SMT* envoyées au solveur *SMT*. Nous procédons de la même manière pour `angr` et son module Claripy. Nous effectuons la réécriture des expressions telle que décrite précédemment. Nous obtenons alors des formules *SMT* contenant un nombre de variables et de contraintes plus important que les formules *SMT* originales. Nous forçons ensuite le moteur à résoudre ces nouvelles formules uniquement de manières incrémentales. Nous présentons dans la figure 6.1 un exemple des mesures du taux de croissance temporel aux appels au SAT-solveur pour la résolution des requêtes *SMT* pour différentes fonctions `APhash` (32-bits) avec différentes tailles d'entrées, sans modifications des requêtes *SMT*. Nous pouvons observer des comportements différents pour les différentes entrées, mais aussi le fait que ce taux, en valeur absolue, dépasse bien la valeur de 69% .
Les résultats de ces expériences se trouvent dans le tableau 6.2, pour une entrée de 40 lettres. Nos mesures de temps sont approximatives.

Nous précisons que les mesures de temps s'effectuent de la manière suivante. Nous mesurons le temps d'analyse effectué par le moteur jusqu'à la détection d'un prédicat opaque. Nous ne mesurons pas le temps de redéfinition de la requête. Nous mesurons le temps de résolution de

6. `http://www.informatik.uni-bremen.de/agra/eng/metasmt.php`
7. basé sur LLVM

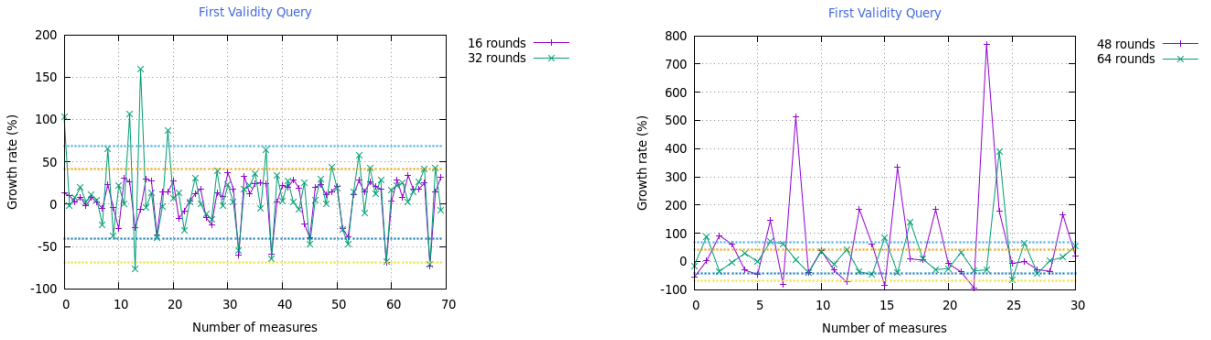


FIGURE 6.1 – Avec KLEE-Z3, mesures du taux de croissance temporel des appels au SAT-solveur pour la fonction APhash (32-bits), pour des entrées de 16, 32, 48, 64 lettres

Programmes	KLEE-Boolector			KLEE-Z3			Angr-Z3		
	Détection	Temps d'analyse		Détection	Temps d'analyse		Détection	Temps d'analyse	
		Avant	Après		Avant	Après		Avant	Après
APhash (16-bits)	1/1	1 260	311	1/1	T.O.	90	1/1	1 021	180
APhash (32-bits)	1/1	6 600	250	1/1	T.O.	243	1/1	T.O.	562

TABLE 6.2 – Mesures approximatives de la détection de prédicats opaques et des temps d'analyse de programmes contenant des prédicats opaques, avant et après la redéfinition des requêtes *SMT*. (Temps en secondes)

l'ancienne requête (s'il est inférieur à 2 heures, sinon nous notons T.O. (*durée dépassée* (*Time Out*)), et le(s) temps de résolution de(s) nouvelle(s) requête(s). Puis, nous mesurons, si cela est possible, le temps nécessaire pour réaliser le reste de l'analyse si d'autres prédicats opaques ne sont pas rencontrés, sinon nous recommençons nos mesures de temps. Le Temps d'analyse Avant correspond à la somme des temps mesurés sans modification des requêtes générés par le moteur *DSE*. Le Temps d'analyse Après correspond à la somme des temps mesurés avec modifications des requêtes.

6.5 Travaux existants

L'utilisation d'un moteur *DSE* pour l'analyse et la détection des prédicats opaques est une technique récente et semble être non adaptée à tous les contextes d'analyse [Oll+19a]. Pour pallier cette difficulté, les solutions actuelles proposent d'adapter la stratégie utilisée par un *DSE*, ou d'ajouter des modules dédiés exclusivement à l'analyse des prédicats opaques. TOFIGHI-SHIRAZI *et al.* [Tof+19] proposent l'utilisation d'une technique de *machine learning* pour détecter des prédicats opaques, afin de les remplacer par des constantes durant une analyse statique. D'un

point de vue plus formel, il n'existe pas à notre connaissance de travaux dédiés à l'analyse des prédicats opaques dans une théorie, mais des travaux plus généraux sur les structures finies, comme l'explique KOPONEN [Kop11]. Malheureusement, peu de ces résultats sont exploités en informatique.

6.6 Conclusion du chapitre

Dans ce chapitre, l'observation des prédicats opaques nous a permis de comprendre qu'il existe des structures donnant lieu à plusieurs interprétations possibles, lorsqu'elles sont exprimées dans une théorie ω-catégorique. Comme nos expériences le montrent, ces différentes interprétations ne possèdent pas le même temps d'analyse. Nous pouvons alors répondre positivement à notre première question de recherche **QR3** : *possède-t-on un avantage temporel lorsque des règles de redéfinitions sont utilisées sur une requête SMT contenant un prédicat opaque ?* Nous montrons aussi que le problème de la réduction de l'opacité des prédicats opaques revient à essayer de comprendre comment les modèles d'un prédicat sont calculables et calculés. Autrement dit, l'interprétation de ces structures automatiques par un moteur d'exécution symbolique dynamique révèle que ces structures nécessitent un nombre important d'étapes de décomposition. Comme nous ne savons pas précisément comment tous les prédicats opaques sont construits, nous ne pouvons pas dire précisément si l'ajout des contraintes, et/ou de création des sous-requêtes permet de réduire le temps d'analyse de tous les prédicats opaques. Nous pouvons juste observer par nos expériences que cette dernière fonctionne dans de nombreux cas, ce qui nous indique que notre idée peut être généralisée à d'autres outils d'analyse automatique tels que ceux analysant des binaires (par exemple : `S2E` [CKC11], `BINSEC` [Dav+16], `angr` [Sho+16], `Triton` [SS15], *etc.*). Une piste future pourrait être la création d'un générateur de prédicat opaque qui puisse proposer différentes représentations pour chaque prédicat. Nous pourrions alors étudier plus facilement l'impact des prédicats opaques dans une analyse statique.

Chapitre 7

Participation à la compétition SMT-COMP

Sommaire

7.1 Introduction 145
7.2 Requêtes *SMT* 146
7.3 Résultats de la compétition et interprétations 152
7.4 Conclusion du chapitre 155

Résumé À notre connaissance, il n'existe pas de bases de données de prédicats opaques accessible librement, et encore moins de requêtes *SMT* contenant spécifiquement des prédicats opaques. C'est pourquoi nous avons soumis neuf requêtes *SMT* de ce type, à la compétition SMT-COMP 2019. Cette compétition confronte les meilleurs solveurs *SMT* face à différents challenges. Chaque requête définie dans la théorie QF-AUFBV, a été construite à partir de programmes contenant chacun un prédicat opaque, qui ne possède pas de problème difficile. Notre objectif était d'obtenir des requêtes demandant un temps de résolution exagérément long pour être décidables. Les résultats obtenus montrent que quatre de ces requêtes ne peuvent être résolues dans le temps imparti. Ce travail a été réalisé avec l'aide de Olivier DECOURBE (Inria), et de Sébastien JOSSE (Dga).

7.1 Introduction

Afin d'affiner la déobfuscation de programmes protégés par des prédicats opaques, nous avons souhaité mettre à disposition des requêtes *SMT* contenant des prédicats opaques, et ne contenant pas de problèmes difficiles. Proposer un tel ensemble de requêtes permettrait à la communauté sécurité informatique et à la communauté *SMT* d'avoir des exemples concrets de requêtes contenant des prédicats opaques. Ainsi, cet ensemble de requêtes doit répondre à

l'objectif suivant : *des requêtes sont difficilement résolubles pour les meilleurs solveurs SMT*. Pour être accessibles au plus grand nombre, nous avons souhaité insérer ces requêtes dans la bibliothèque *SMT-LIB* [1]. Dans ce but, nous avons soumis nos requêtes à la compétition SMT-COMP [19a], organisée par la communauté *SMT*. Cette compétition confronte les meilleurs solveurs *SMT* à des *étalons* (*benchmarks*) pour les comparer. La compétition était classée en différentes catégories pour l'année 2019, qui se différenciaient principalement sur le temps d'analyse des requêtes, et sur la nature des requêtes. Elle se déroula dans le cluster `Starexec` [2], financée par la *National Science Fundation* [3]. Cette plate-forme est aussi utilisée par d'autres communautés comme *TPTP* [4]. Le cluster se compose de 192 machines équipées de deux processeurs CPU Intel® Xeon™ E5-2609 (10 MB Cache, 2.4 GHz). La configuration est restée la même depuis le début de la compétition en 2013. Nous avons soumis nos requêtes à la catégorie *industrial*, qui acceptait des requêtes non incrémentales de la théorie QF-AUFBV, avec un *temps limite* (*Time out*) de résolution fixé à 12 heures (43 200 secondes). Chacun de ces processeurs possède 4 cœurs avec 64 Go de *RAM*.

Nous décrivons dans la suite de ce chapitre, les requêtes *SMT* soumises et les résultats des résolutions obtenues.

7.2 Requêtes *SMT*

Dans cette section, nous présentons les requêtes *SMT*, ainsi qu'un exemple de construction de requête.

7.2.1 Description

Nous avons soumis 9 requêtes, fabriquées à partir de trois types de programmes contenant au choix :

- *type 1* : un prédicat composé à l'aide d'une fonction algébrique,
- *type 2* : un prédicat appartenant à la famille *Mixed Boolean Arithmetic* (*MBA*),
- *type 3* : un prédicat mettant en relation deux polynômes multivariés.

1. `https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_AUFBV/tree/master/2019-Gonzalvez`
2. `https://www.starexec.org`
3. `https://www.nsf.gov/`
4. *TPTP* : *Thousands of Problems for Theorem Provers*

Le comportement constant de ces prédicats ont été vérifiés par exécution, c'est-à-dire que nous avons exécuté toutes les entrées possibles de chaque programme, avons observé le comportement constant du prédicat, et son exécution rapide. Ces trois types de programmes nous donnent trois types de requêtes. Une description plus précise du premier type de requête (`opStructure_O`* et `opStructure_C`*) est donnée dans la sous-section suivante. Les requêtes contenant des *MBA* (`opStructure_MBA_6` et `opStructure_MBA_7`) ont été fournies par Sébastien JOSSE, et créées à partir du générateur de requêtes ayant servi à la publication [Bio+17]. Nous ne décrirons pas ici le fonctionnement des *MBA*, mais nous retenons seulement que le comportement des *MBA* peut être capturé dans (ou composé avec) l'aide de la théorie QF-BV. Les caractéristiques de nos *MBA* sont les suivantes : degrés du polynôme 6 (pour la requête `opStructure_MBA_6`) ou 7 (pour la requête `opStructure_MBA_7`), mots de 32-bits et expressions *MBA* 4×4. Pour le dernier type de requête (`opStructure_NPT_1` et `opStructure_NPT_2`), nous comparons la valeur de deux fonctions polynomiales. Le schéma de construction est similaire à celui décrit dans le listing 7.2, sans les lignes 11 et 21, avec un nombre de tours de boucle fixé.

Ajoutons enfin que pour obtenir les requêtes du premier et troisième type, nous avons délégué cette tâche à `KLEE` de la manière suivante : nous soumettons un code source C contenant un prédicat opaque à `KLEE` [5] utilisant le solveur `Z3`, et nous lui demandons de nous retourner toutes les requêtes *SMT* fabriquées, le résultat et le temps de chaque résolution. Nous avons observé que `KLEE` fabrique des séquences de requêtes *SMT* pour résoudre la condition de branchement contenant le prédicat opaque. Nous avons extrait pour chaque séquence la requête dont le temps de résolution est le plus grand. Le tableau 7.1 présente les sorties observées pour ces requêtes avec `Z3`, et la taille de ces requêtes. Le listing 7.1 présente un exemple de requête soumise à la compétition.

7.2.2 Description de la construction du premier type de requêtes

Nous décrivons ici les principales étapes réalisées pour construire le premier type de requêtes. Nous utilisons la fonction `Predicate` décrite dans le listing 7.2. Cette fonction accepte en entrée une variable entière non signée, et retourne *quasiment* [6] toujours la valeur 0. Les entrées permettant d'obtenir des valeurs différentes de 0 ne sont pas consécutives. Pour obtenir ce comportement, la fonction `Predicate` fait appel à la fonction `Func_1` qui accepte en en-

5. Nous avons activé l'option *Static single assignment* (*SSA*) pour la représentation intermédiaire `LLVM`, principalement pour étaler les boucles contenues dans le programme.

6. *quasiment* dans le sens à quelques cas près

Noms	Résultats	Taille (en kilo-octets)
opStructure_O_16_24_2_2	Unknown	134,5
opStructure_O_16_32_2_2	Unknown	179,6
opStructure_O_32_16_2_2	Unknown	177,4
opStructure_O_32_32_2_2	Unknown	359,2
opStructure_C_16_32_2_2	UNSAT	40,20
opStructure_MBA_6	SAT	739,6
opStructure_MBA_7	SAT	2100
opStructure_NPT_1	SAT	2,7
opStructure_NPT_2	SAT	13,3

TABLE 7.1 – Description courte des requêtes *SMT*

trée une variable entière non signée, et qui retourne *quasiment* [7] une fois sur deux la valeur 0. Quatre variables, notées dans le programme `VAR_A`, `VAR_B`, `VAR_C`, `VAR_D`, permettent d'obtenir différente version du programme. Ces variables se retrouvent dans les noms des requêtes (`opStructure_*_{VAR_A}_{VAR_B}_{VAR_C}_{VAR_D}`). Deux prédicats sont utilisés dans ce premier type, qui sont `Predicate(x) != Predicate(x+1)`, et `Predicate(x) != X`, où `X` est une constante. Le premier prédicat correspond à la lettre `O` dans le nom des requêtes, et le second à la lettre `C`.

L'idée principale pour construire ces fonctions consiste à créer un polynôme où chaque opérande est un représentant d'un groupe abélien. Ces représentants sont définis à l'aide de cycles de permutations, calculées à partir d'un produit externe [8], appelé *produit couronne externe* (*external wreath product*) [9], et d'un *produit du groupe interne*, appelé *produit cartésien* (*cartesian product*). Dit plus simplement, chaque opérande est composé à l'aide de deux multiplications ne possédant pas les mêmes propriétés. Il n'existe pas à notre connaissance d'outils informatiques permettant d'utiliser les *produits couronnes* pour obtenir des cycles de permutations. Nous avons calculé manuellement des cycles de permutations, à partir des travaux de IBRAHIM et AUDU [IA07]. Puis, après avoir choisi deux cycles de permutation au hasard, nous les avons représentés approximativement avec des symboles du langage C (c'est-à-dire avec des variables, des opérateurs arithmétiques, et des opérateurs booléens). Le produit cartésien de ces deux

7. *quasiment* dans le sens $50\% \pm 5\%$

8. Ici, on parle bien d'un *produit de groupe externe*, à ne pas confondre avec le *produit externe des espaces vectoriels*.

9. Ce produit permet de construire des groupes à partir d'autres groupes.

```
(set-info :smt-lib-version 2.6)
(set-logic QF_AUFBV)
; This file was generated with the help of KLEE
(set-info :source |
Generated by: Alexandre Gonzalvez
Generated on: 2019-02-28
Generator: KLEE
Target solver: Boolector Z3 STP

Benchmarks arising from a generator of opaque expressions
These expressions are used to protect some constants against
reverse-engineering, but their robustness can be compromised by
SMT-solvers
|)
(set-info :license "https://creativecommons.org/licenses/by/4.0/")
(set-info :category "industrial")
(set-info :status sat)
; Number of instructions: 475
; Time expected : less than 1 minute

(declare-fun model_version () (Array (_ BitVec 32) (_ BitVec 8) ) )
(declare-fun x () (Array (_ BitVec 32) (_ BitVec 8) ) )
(assert (let ( (?B1 (concat (select x (_ bv3 32) ) (concat (select x (_ bv2 32) ) (concat (↩
    select x (_ bv1 32) ) (select x (_ bv0 32) ) ) ) ) ) ) (let ( (?B2 (bvmul ?B1 ?B1 ) ) ) (↩
    let ( (?B3 (bvmul (_ bv715827882 32) ?B2 ) ) ) (let ( (?B4 (bvadd (bvmul ?B2 (bvlshr (↩
    bvadd (_ bv2863311530 32) ?B3 ) (_ bv2 32) ) ) ?B3 ) ) ) (let ( (?B5 (bvadd (bvmul ?B2 (↩
    bvlshr (bvadd (_ bv2863311530 32) ?B4 ) (_ bv2 32) ) ) ?B4 ) ) ) (let ( (?B6 (bvadd (↩
    bvmul ?B2 (bvlshr (bvadd (_ bv2863311530 32) ?B5 ) (_ bv2 32) ) ) ?B5 ) ) ) (let ( (?B7 (↩
    bvadd (bvmul ?B2 (bvlshr (bvadd (_ bv2863311530 32) ?B6 ) (_ bv2 32) ) ) ?B6 ) ) ) (let (↩
     (?B8 (bvadd (bvmul ?B2 (bvlshr (bvadd (_ bv2863311530 32) ?B7 ) (_ bv2 32) ) ) ?B7 ) ) )↩
     (let ( (?B9 (bvadd (bvmul ?B2 (bvlshr (bvadd (_ bv2863311530 32) ?B8 ) (_ bv2 32) ) ) ?↩
    B8 ) ) ) (let ( (?B10 (bvadd (bvmul ?B2 (bvlshr (bvadd (_ bv2863311530 32) ?B9 ) (_ bv2 ↩
    32) ) ) ?B9 ) ) ) (let ( (?B11 (bvadd (bvmul ?B2 (bvlshr (bvadd (_ bv2863311530 32) ?B10 ↩
    ) (_ bv2 32) ) ) ?B10 ) ) ) (let ( (?B12 (bvadd (bvmul ?B2 (bvlshr (bvadd (_ bv2863311530↩
     32) ?B11 ) (_ bv2 32) ) ) ?B11 ) ) ) (let ( (?B13 (bvadd (bvmul ?B2 (bvlshr (bvadd (_ ↩
    bv2863311530 32) ?B12 ) (_ bv2 32) ) ) ?B12 ) ) ) (let ( (?B14 (bvadd (bvmul ?B2 (bvlshr ↩
    (bvadd (_ bv2863311530 32) ?B13 ) (_ bv2 32) ) ) ?B13 ) ) ) (let ( (?B15 (bvadd (bvmul ?↩
    B2 (bvlshr (bvadd (_ bv2863311530 32) ?B14 ) (_ bv2 32) ) ) ?B14 ) ) ) (let ( (?B16 (↩
    bvadd (bvmul ?B2 (bvlshr (bvadd (_ bv2863311530 32) ?B15 ) (_ bv2 32) ) ) ?B15 ) ) ) (let↩
     ( (?B17 (bvadd (bvmul ?B2 (bvlshr (bvadd (_ bv2863311530 32) ?B16 ) (_ bv2 32) ) ) ?B16 ↩
    ) ) ) (and (= (_ bv3657258328 32) (bvadd (bvmul ?B2 (bvlshr (bvadd (_ bv2863311530 32) ?↩
    B17 ) (_ bv2 32) ) ) ?B17 ) ) (= (_ bv1 32) (concat (select model_version (_ bv3 32) ) (↩
    concat (select model_version (_ bv2 32) ) (concat (select model_version (_ bv1 32) ) (↩
    select model_version (_ bv0 32) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) )
(check-sat)
(exit)
```

Listing 7.1 – Requête *SMT*: `opStructure_NPT_1.smt2`

cycles nous donne un représentant d'un groupe abélien. Ainsi, pour toutes les valeurs d'entrées acceptées par cette expression, les opérations effectuées sont géométriquement définies dans un groupe commutatif. Dans notre exemple (listing 7.2), l'expression finale se situe ligne 11. Le premier cycle est (234561) et correspond à `((var_0 * 2) + ((var_0 & 256) / 256))`. Le second cycle est (612345) et correspond à `(((var_0 / 2) ^ ((var_0 & 64) / 16) + ((var_0 & 9) * 32 )))`.

Plus formellement, nous avons combiné des connaissances en théorie des groupes, en théorie de la représentation et en théorie des modèles. En théorie des groupes, nous nous sommes appuyés sur les résultats concernant l'équivalence qui concerne l'*action de groupe* [10]*produit* (*product action*) [CGM08] entre le *produit cartésien* et le *produit couronne externe*. En effet, dans le cas de l'*action* d'un *produit couronne* sur un *produit cartésien* [11], cette *action* ne peut être réalisée qu'à la condition de l'existence d'une équivalence entre le *produit semi-direct* [12] *interne* et *externe* des ensembles manipulés. Dit plus simplement, cette *action* est basée sur l'équivalence existante entre les deux situations suivantes : « deux groupes plongés dans un troisième » et « deux groupes non *a priori* plongés dans un troisième ». De plus, des résultats en théorie de la représentation montrent que cette équivalence n'est pas toujours linéaire [13], notamment pour l'action d'un groupe sur un espace vectoriel [14]. Or, comme nous l'avons vu dans l'exemple 16 (chapitre 5), pour la théorie QF-BV, les éléments de cette théorie sont définis dans le *groupe linéaire affine* AGL, c'est-à-dire définis dans un produit *semi-direct extérieur* entre un groupe et un espace vectoriel. De ces constats, nous avons supposé que cette condition d'équivalence est implicite lors de la résolution d'une requête définie dans la théorie QF-BV. Dit autrement, nous avons supposé que cette condition est un aspect implicite du problème *SMT*, pouvant être exploitée par un prédicat opaque.

10. Une action de groupe sur un ensemble est une loi de composition externe d'un groupe sur un ensemble, vérifiant des conditions supplémentaires.

11. `https://groupprops.subwiki.org/wiki/Action_of_wreath_product_on_Cartesian_product`

12. Le *produit semi-direct* permet de définir un groupe G à partir de deux groupes A et B.

13. *cf.* notes du cours de Constantin Teleman sur la théorie de la représentation : `https://math.berkeley.edu/~teleman/math/RepThry.pdf`

14. `https://math.stackexchange.com/questions/1445825/example-of-a-group-action-g-on-a-vector-space-v-that-fails-to-be-linear-i-e-fa`

```
#define VAR_A 16
#define VAR_B 32
#define VAR_C 2
#define VAR_D 2

unsigned int Func_1(unsigned int in1){
 unsigned int var_0, i;
 var_0 = 0xAAAAAAAA;
 for(i=0; i < VAR_A; i++){
   var_0 += in1 * in1 * (var_0 >> VAR_C); /* circuit unique avec un carre */
   var_0 = ((var_0*2) + ((var_0&256) /256) ) * (((var_0 /2) ^ ((var_0&64)/16) + ((var_0&9)
    *32))); /* produit cartesien de deux cycles definis via un produit couronne sur un
    ensemble fini de taille 6 bits*/
 }
 return var_0;
}

unsigned int Predicate(unsigned int in2){
 unsigned int var_0, i;
 var_0 = 0xAAAAAAAA;
 for(i=0; i < VAR_B; i++){
   var_0 += in2 * in2 * (var_0 >> VAR_D); /* circuit unique avec un carre */
   var_0 = Func_1(var_0);
 }
 return var_0;
}
```

Listing 7.2 – Fonctions principales utilisées pour fabriquer des prédicats opaques

7.3 Résultats de la compétition et interprétations

Nous retranscrivons maintenant les résultats de la compétition de l'année 2019. L'ensemble de ces résultats peuvent être retrouvés dans le dépôt github de la compétition [15]. Pour chaque requête et chaque solveur, les résultats sont présentés dans 4 tableaux, permettant de présenter :

- le comportement de chaque solveur (tableau 7.2) stipulant la cause de l'arrêt de la résolution (*complete* : la procédure de décision a abouti ; *T.O.* : la procédure de décision n'a pas pu aboutir, car elle dépasse le temps de résolution autorisé ; *memout* : la procédure de décision n'a pas pu aboutir, car la quantité de ressources disponible n'est pas suffisante)

- le résultat obtenu (tableau 7.3) (SAT : la requête est satisfiable ; UNSAT : la requête n'est pas satisfiable ; Unknown : pas de prise de décision sur la satisfiabilité de la requête)

- le temps CPU utilisé (tableau 7.4)

- la quantité de mémoire *RAM* utilisée (tableau 7.5)

Les résultats obtenus lors de la compétition indiquent le comportement attendu décrit dans le tableau 7.1. Les résolutions du premier type de requêtes restent encore trop longues temporellement pour être résolues. Nous pouvons noter aussi que la quantité de mémoire maximale consommée n'atteint pas les limites de mémoire disponible. Les résolutions du deuxième et troisième type nous permettent de constater la faisabilité d'obtenir des requêtes difficiles à résoudre pour certains solveurs et pas pour d'autres. Nous pouvons constater que nos requêtes illustrent bien des cas pratiques de prédicats opaques contenus dans des requêtes *SMT*. Ces requêtes ne semblent pas contenir de problèmes difficiles en apparence, et elles consomment une quantité de mémoire limitée. De plus, nous pouvons aussi constater que pour une même théorie, des requêtes *SMT* peuvent être difficiles à résoudre pour certaines stratégies de résolution et pas pour d'autres. Ainsi, de ces résultats, nous pouvons déduire que l'opacité d'un prédicat dépend bien du choix de sa représentation et de son interprétation. Nous déduisons aussi que le choix de la théorie utilisée lors d'une analyse automatique permet de capturer, ou non, le comportement opaque. Au final, ces résultats nous ont permis d'explorer quelques mécanismes aidant à la réalisation de prédicats opaques.

15. `https://github.com/SMT-COMP/smt-comp/blob/master/2019/results/Challenge_Track_non-incremental.csv.tar.xz`

Requêtes	Boolector	CVC4	Poolector	Yices	Z3
opStructure_0_16_24_2_2	T.O.	complete	T.O.	T.O.	T.O.
opStructure_0_16_32_2_2	T.O.	complete	T.O.	T.O.	T.O.
opStructure_0_32_16_2_2	T.O.	complete	T.O.	T.O.	T.O.
opStructure_0_32_32_2_2	T.O.	complete	T.O.	T.O.	complete
opStructure_C_16_32_2_2	complete	complete	complete	complete	complete
opStructure_MBA_6	complete	complete	complete	complete	complete
opStructure_MBA_7	complete	complete	memout	complete	complete
opStructure_NPT_1	complete	T.O.	complete	T.O.	complete
opStructure_NPT_2	complete	complete	complete	complete	complete

TABLE 7.2 – Comportements observés

Requêtes	Boolector	CVC4	Poolector	Yices	Z3
opStructure_0_16_24_2_2	Unknown	Unknown	Unknown	Unknown	Unknown
opStructure_0_16_32_2_2	Unknown	Unknown	Unknown	Unknown	Unknown
opStructure_0_32_16_2_2	Unknown	Unknown	Unknown	Unknown	Unknown
opStructure_0_32_32_2_2	Unknown	Unknown	Unknown	Unknown	Unknown
opStructure_C_16_32_2_2	UNSAT	UNSAT	UNSAT	UNSAT	UNSAT
opStructure_MBA_6	SAT	SAT	SAT	SAT	SAT
opStructure_MBA_7	SAT	Unknown	Unknown	SAT	Unknown
opStructure_NPT_1	SAT	Unknown	SAT	Unknown	SAT
opStructure_NPT_2	SAT	SAT	SAT	SAT	SAT

TABLE 7.3 – Résultats observés

Requêtes	Boolector	CVC4	Poolector	Yices	Z3
opStructure_0_16_24_2_2	43 197,2	8 249,86	171 253	43 201,5	43 191,2
opStructure_0_16_32_2_2	43 196,3	8 648,55	171 233	43 200,2	43 191,1
opStructure_0_32_16_2_2	43 198,9	9 601,54	171 191	43 196,7	43 193,1
opStructure_0_32_32_2_2	43 195,7	11 514,0	171 258	43 198,6	4 931,38
opStructure_C_16_32_2_2	34,1773	18,6448	59,6200	17,5816	19,2919
opStructure_MBA_6	102,379	11 114,1	448,940	4 555,95	5 041,28
opStructure_MBA_7	218,312	2 629,31	310,060	0,349 048	5 778,00
opStructure_NPT_1	30,7294	43 181,0	169,940	43 195,7	1 374,90
opStructure_NPT_2	0,140 914	296,960	0,458 904	0,035 704	0,624 340

TABLE 7.4 – Temps CPU (en secondes)

Requêtes	Boolector	CVC4	Poolector	Yices	Z3
opStructure_0_16_24_2_2	2 542,2	18 916	10 931	3 784,8	29 300
opStructure_0_16_32_2_2	3 060,5	19 562	13 128	3 512,6	39 657
opStructure_0_32_16_2_2	3 142,3	19 470	12 908	3 722,4	36 042
opStructure_0_32_32_2_2	5 762,3	22 134	24 928	4 182,6	51 257
opStructure_C_16_32_2_2	871,7	940,86	3 467,2	382,01	578,88
opStructure_MBA_6	13 771	47 937	47 251	30 150	40 404
opStructure_MBA_7	30 801	52 742	63 042	110,52	51 586
opStructure_NPT_1	275,70	16 853	2 074,4	523,36	434,13
opStructure_NPT_2	110,52	1 300,8	110,52	110,52	253,63

TABLE 7.5 – Quantité de mémoire utilisée (en méga-octets)

7.4 Conclusion du chapitre

Dans ce chapitre, nous proposons à la communauté *SMT*, au travers de la compétition SMT-COMP 2019, des requêtes contenant des prédicats opaques. Nous avons souhaité par cette occasion affiner nos connaissances sur la déobfuscation de programmes protégés par des prédicats opaques. Nous nous sommes fixé l'objectif suivant : *proposer des requêtes SMT contenant des prédicats opaques ne pouvant être résolus par des solveurs SMT*. Nous souhaitons aussi faciliter l'étude des prédicats opaques contenues dans des requêtes *SMT*. Des neuf requêtes que nous avons soumises, quatre ne sont pas résolubles par les solveurs *SMT*. Les résultats de la compétition nous ont permis d'observer que : l'opacité d'un prédicat s'exprime en fonction du choix de la représentation et de l'interprétation. Nous avons vu aussi dans ces résultats que le temps de résolution d'une requête *SMT* peut beaucoup varier en fonction du solveur utilisé. Au moment de la rédaction de ce manuscrit, nos requêtes ont été réutilisées pour la compétition SMT-COMP 2020 [16], avec d'autres solveurs, et nous attendons de voir ces nouveaux résultats. Une piste future pour mieux comprendre les effets de nos requêtes serait de formaliser en détail ce qui modifie les temps de résolution en fonction des procédures de décision et des stratégies utilisées. Ceci permettrait d'améliorer le fonctionnement des solveurs *SMT*, et la compréhension des mécanismes de déobfuscation. Une autre piste pourrait être de prolonger ce travail pour d'autres théories, en prenant en compte les travaux formels de Bodirsky et Bodor [BB18] sur des structures dont le comportement ressemble fortement au comportement attendu des prédicats opaques.

16. `https://github.com/SMT-COMP/smt-comp/blob/master/2020/prep/selection/final/benchmark_selection_single_query_2020.tar.xz`

Troisième partie

Conclusion générale

Chapitre 8

Conclusion générale

8.1 Résumé des contributions

Dans ce manuscrit, nous avons su relever un certain nombre de défis, et mettre en lumière un certain nombre de limites.

Pour étudier un cas réaliste, nous avons choisi de nous concentrer sur l'analyse statique de programmes composés de prédicats opaques. Une telle analyse peut être réalisée à l'aide d'un moteur d'exécution symbolique dynamique, qui permet de déobfusquer automatiquement un programme, et qui utilise un solveur *SMT*. Nous avons regardé le fonctionnement de ce moteur, ainsi que du solveur. Nous avons observé un des effets d'un prédicat opaque sur ces outils.

Nous avons ensuite regardé dans une machine simple, comment l'instruction de saut indirect permet de garantir, ou non, l'intégrité matérielle du flot de contrôle d'un programme. Les prédicats opaques obfusquent les instructions de sauts indirects. Nous montrons que garantir l'intégrité d'un programme n'est pas toujours compatible avec une volonté de protéger un programme par obfuscation.

Nous poursuivons notre travail sur l'analyse de programmes contenant des prédicats opaques. Nous proposons la conjecture suivante sur la structure des prédicats opaques : *L'interprétation d'un prédicat opaque dans une théorie homogène recouvre une structure automatique qui possède la propriété d'indépendance.* Nous proposons d'adapter un moteur *DSE* et plusieurs solveurs *SMT* à la détection d'une structure possédant une propriété d'indépendance. Nous testons et détectons des prédicats opaques avec cette méthode.

Nous déduisons de notre conjecture que l'interprétation d'un prédicat opaque peut ne pas être stable, et que cette instabilité est la source de l'opacité. Pour réduire cette opacité, nous proposons une méthode de redéfinition d'une requête *SMT* et d'ordonnancement de sous-requêtes *SMT* à résoudre. Nous observons une réduction du temps d'analyse de programmes obfusqués.

Enfin, nous avons soumis à la compétition SMT-COMP 2019 quelques requêtes *SMT* (pour la théorie QF-AUFBV) contenant des prédicats opaques que nous avons fabriqué. Ces requêtes ont été acceptées, et sont maintenant utilisables librement. La compétition SMT-COMP permet

d'alimenter une base de données de requêtes à destination principale de la communauté *SMT*. Les requêtes de cette base de données sont principalement utilisées pour comparer les solveurs *SMT*.

8.2 Perspectives

En s'appuyant sur ces contributions, trois pistes peuvent être explorées pour approfondir ce travail :

- caractériser la difficulté à interpréter des familles de prédicats opaques en fonction de la théorie utilisée ;
- extraire automatiquement les prédicats opaques contenus dans des programmes ;
- réaliser un générateur de prédicats opaques.

8.2.1 Caractérisation des prédicats opaques en fonction d'une théorie

La première orientation concerne l'approfondissement de notre conjecture. À notre connaissance, il n'existe pas de méthode permettant de savoir quand et où des prédicats opaques peuvent être utilisés de manière judicieuse et efficace en tant que méthode d'obfuscation. Cela peut être particulièrement intéressant dans la situation décrite ci-après. Un prédicat opaque est utilisé pour protéger des programmes contre un attaquant ne sachant pas comment interpréter ce prédicat. Si un attaquant connaît une méthode d'interprétation efficace, la protection recherchée ne semble plus faire son effet. Il nous paraît tout à fait possible de comprendre comment cette protection est modulée en fonction de la méthode d'interprétation utilisée. Malheureusement, les paramètres pouvant intervenir dans ce cas d'étude sont multiples : choix de la méthode de raisonnement, choix de la méthode d'interprétation, choix du modèle de calculabilité, *etc.* Un point de départ potentiellement intéressant est d'analyser automatiquement des membres de différentes familles de prédicats opaques, à l'aide d'un système de preuve fixé (ou une méthode d'analyse statique fixée) pour une théorie décidable fixée. Puis de recommencer avec les mêmes prédicats et le même système d'analyse, mais avec une autre théorie décidable. Si les effets des prédicats opaques sont observés avec des théories décidables différentes et n'ayant pas les mêmes propriétés, alors ce serait une preuve supplémentaire pour notre conjecture. Connaître ces caractéristiques permettrait aussi d'améliorer le fonctionnement des solveurs *SMT*, notamment pour de l'analyse de fonctions cryptographiques, comme le montre les travaux de Nejati [Nej20].

8.2.2 Extraction automatique de prédicats opaques

La deuxième orientation concerne la continuité des travaux de détection des prédicats opaques. Une fois un prédicat opaque découvert, les extraire de leur environnement original n'est pas toujours chose facile, car le prédicat peut être composé avec d'autres méthodes de protections. Une extraction automatique faciliterait la mise en place d'une base de données exploitable pour d'autres applications. La difficulté dans ce cas est de choisir une méthode permettant d'extraire un large éventail de prédicats opaques connus et inconnus. Ajoutons que la quantité d'informations minimale nécessaires devant être extraites d'un programme pour qu'un prédicat opaque puisse agir n'est toujours pas connue. Il nous semble qu'un travail de fond sur les structures qui peuvent être interprétées comme des prédicats opaques permettrait d'éclaircir ce qui peut être extrait automatiquement de ce qui ne l'est pas. Une application directe et simple de cette orientation serait d'extraire automatiquement les prédicats opaques contenus dans des logiciels malveillants.

8.2.3 Réalisation d'un générateur de prédicats opaques

La troisième orientation concerne la génération de prédicats opaques. À notre connaissance, il n'existe pas de méthode robuste permettant de réaliser des prédicats opaques. Un générateur de prédicats opaques permettrait de simplifier l'étude des prédicats opaques sur plusieurs points encore obscurs de nos jours : formes, propriétés, interprétations. Il deviendrait facile de tester les outils d'analyse statique face aux prédicats opaques et inversement. Cela permettrait aussi d'isoler facilement les prédicats opaques difficiles à analyser, et de mieux comprendre formellement comment l'obfuscation est réalisable dans un programme. Par extension, cette orientation pourrait aider à une meilleure compréhension de l'obfuscation opérationnelle dans de vrais programmes, ce qui apporterait aussi des pistes sérieuses pour le développement de l'obfuscation cryptographique.

Bibliographie

[18] *Clang : a C language family frontend for LLVM*, http://releases.llvm.org/, Accessed : 2019-09-01, 2018.

[19a] *The SMT-COMP*, https://smt-comp.github.io/2019/index.html, Accessed : 2019-09-01, 2019.

[19b] *Z3 Tutorial*, https://theory.stanford.edu/~nikolaj/programmingz3.html, Accessed : 2019-09-01, 2019.

[20a] *ASProtect*, http://www.aspack.com/asprotect32.html, 2020.

[20b] *PELock software*, https://www.pelock.com/, 2020.

[20c] *Vmprotect Software. Vmprotect.* http://vmpsoft.com, 2020.

[Aba+09] Martín Abadi *et al.*, « Control-flow integrity principles, implementations, and applications », in : *ACM Transactions on Information and System Security* 13.*1* (1er oct. 2009), p. 1-40, ISSN : 10949224, DOI : 10.1145/1609956.1609960, URL : http://portal.acm.org/citation.cfm?doid=1609956.1609960 (visité le 23/07/2018).

[Akh+15] Adnan Akhunzada *et al.*, « Man-At-The-End attacks : Analysis, taxonomy, human aspects, motivation and future directions », in : *Journal of Network and Computer Applications* 48 (2015), p. 44-57.

[Anc+07] Bertrand Anckaert *et al.*, « Program obfuscation : a quantitative approach », in : *Proceedings of the 2007 ACM workshop on Quality of protection*, ACM, 2007, p. 15-20.

[Apo+14] Daniel Apon *et al.*, « Implementing Cryptographic Program Obfuscation. », in : *IACR Cryptology ePrint Archive* 2014 (2014), p. 779.

[App02] Andrew Appel, « Deobfuscation is in NP », in : *Princeton University, Aug* 21 (2002), p. 2.

[Ban+13] Julian Bangert *et al.*, « The page-fault weird machine : lessons in instruction-less computation », in : *Presented as part of the 7th {USENIX} Workshop on Offensive Technologies*, 2013.

[Ban+16] Sebastian Banescu *et al.*, « Code obfuscation against symbolic execution attacks », in : *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 2016, p. 189-200.

[Bar+01] Boaz Barak *et al.*, « On the (im) possibility of obfuscating programs », in : *Annual International Cryptology Conference*, Springer, 2001, p. 1-18.

[Bar+14] Boaz Barak *et al.*, « Obfuscation for evasive functions », in : *Theory of Cryptography Conference*, Springer, 2014, p. 26-51.

[Bar+19] James Bartusek *et al.*, « New techniques for obfuscating conjunctions », in : *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2019, p. 636-666.

[Bar01] Boaz Barak, « How to go beyond the black-box simulation barrier », in : *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, IEEE, 2001, p. 106-115.

[BB06] Robert Brummayer et Armin Biere, « Local two-level and-inverter graph minimization without blowup », in : *Proc. MEMICS* 6 (2006), p. 32-38.

[BB18] Manuel Bodirsky et Bertalan Bodor, « Structures with Small Orbit Growth », in : *arXiv preprint arXiv :1810.05657* (2018).

[BBL08] Robert Brummayer, Armin Biere et Florian Lonsing, « BTOR : bit-precise modelling of word-level problems for model checking », in : *Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning*, 2008, p. 33-38.

[BDM17] Sébastien Bardin, Robin David et Jean-Yves Marion, « Backward-bounded DSE : targeting infeasibility questions on obfuscated codes », in : *2017 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2017, p. 633-651.

[BDS02] Clark W. Barrett, David L. Dill et Aaron Stump, « A Generalization of Shostak's Method for Combining Decision Procedures », in : *Proceedings of the 4th International Workshop on Frontiers of Combining Systems (FroCoS '02)*, sous la dir. d'Alessandro Armando, t. 2309, Lecture Notes in Artificial Intelligence, Santa Margherita Ligure, Italy, Springer-Verlag, avr. 2002, p. 132-146, url : `http://www.cs.stanford.edu/~barrett/pubs/BDS02-FROCOS02.pdf`.

[BG00] Achim Blumensath et Erich Gradel, « Automatic structures », in : *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No. 99CB36332)*, IEEE, 2000, p. 51-62.

[BHM09] Armin Biere, Marijn Heule et Hans van Maaren, *Handbook of satisfiability*, t. 185, IOS press, 2009.

[BHW11] Armin Biere, Keijo Heljanko et Siert Wieringa, *AIGER 1.9 And Beyond*, rapp. tech. 11/2, Altenbergerstr. 69, 4040 Linz, Austria : Institute for Formal Models et Verification, Johannes Kepler University, 2011.

[Bio+17] Fabrizio Biondi *et al.*, « Effectiveness of synthesis in concolic deobfuscation », in : *Computers & Security* 70 (2017), p. 500-515.

[Bit+13] Nir Bitansky *et al.*, « More on the Impossibility of Virtual-Black-Box Obfuscation with Auxiliary Input. », in : *IACR Cryptology ePrint Archive* 2013 (2013), p. 701.

[BK08] Christel Baier et Joost-Pieter Katoen, *Principles of model checking*, MIT press, 2008.

[BKS15] Bertalan Bodor, Kende Kalina et Csaba Szabó, « Permutation groups containing infinite linear groups and reducts of infinite dimensional linear spaces over the two element field », in : *arXiv preprint arXiv :1506.00220* (2015).

[Bla+17] Tim Blazytko *et al.*, « Syntia : Synthesizing the semantics of obfuscated code », in : *USENIX Security Symposium. Usenix*, 2017.

[BM07] Silvia Barbina et Dugald Macpherson, « Reconstruction of homogeneous relational structures », in : *The Journal of Symbolic Logic* 72.*3* (2007), p. 792-802.

[BMM06] Danilo Bruschi, Lorenzo Martignoni et Mattia Monga, « Detecting self-mutating malware using control-flow graph matching », in : *International conference on detection of intrusions and malware, and vulnerability assessment*, Springer, 2006, p. 129-143.

[BMM07] Danilo Bruschi, Lorenzo Martignoni et Mattia Monga, « Code normalization for self-mutating malware », in : *IEEE Security & Privacy* 5.*2* (2007), p. 46-54.

[BN15] Finn Brunton et Helen Nissenbaum, *Obfuscation : A user's guide for privacy and protest*, Mit Press, 2015.

[Bou70] Nicolas Bourbaki, *Théorie des ensembles*, t. 1, Hermann Paris, 1970.

[BP01] Paul Beame et Toniann Pitassi, « Propositional Proof complexity : Past, Present », in : *Future* (2001), p. 42-70.

[BP18] Sebastian Banescu et Alexander Pretschner, « A tutorial on software obfuscation », in : *Advances in Computers*, t. 108, Elsevier, 2018, p. 283-353.

[BR14a] Zvika Brakerski et Guy N Rothblum, « Black-box obfuscation for d-CNFs », in : *Proceedings of the 5th conference on Innovations in theoretical computer science*, ACM, 2014, p. 235-250.

[BR14b] Zvika Brakerski et Guy N Rothblum, « Virtual black-box obfuscation for all circuits via generic graded encoding », in : *Theory of Cryptography Conference*, Springer, 2014, p. 1-25.

[Bru+19] Pierrick Brunet *et al.*, « Epona and the Obfuscation Paradox : Transparent for Users and Developers, a Pain for Reversers », in : *Proceedings of the 3rd ACM Workshop on Software Protection*, 2019, p. 41-52.

[BS05] Arini Balakrishnan et Chloe Schulze, « Code obfuscation literature survey », in : *CS701 Construction of compilers* 19 (2005).

[BS16] Mihir Bellare et Igors Stepanovs, « Point-function obfuscation : a framework and generic constructions », in : *Theory of Cryptography Conference*, Springer, 2016, p. 565-594.

[BST+10] Clark Barrett, Aaron Stump, Cesare Tinelli *et al.*, « The smt-lib standard : Version 2.0 », in : *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, t. 13, 2010, p. 14.

[BTT+16] Vivek Balachandran, Darell JJ Tan, Vrizlynn LL Thing *et al.*, « Control flow obfuscation for android applications », in : *Computers & Security* 61 (2016), p. 72-93.

[Büc60] J Richard Büchi, « Weak second-order arithmetic and finite automata », in : *Mathematical Logic Quarterly* 6.*1-6* (1960), p. 66-92.

[Bur+17] Nathan Burow *et al.*, « Control-Flow Integrity : Precision, Security, and Performance », in : *ACM Computing Surveys* 50.*1* (4 avr. 2017), p. 1-33, issn : 03600300, doi : 10.1145/3054924, url : http://dl.acm.org/citation.cfm?doid=3058791.3054924 (visité le 20/07/2018).

[Cai+10] Alan J Cain *et al.*, « Automatic presentations and semigroup constructions », in : *Theory of Computing Systems* 47.*2* (2010), p. 568-592.

[Cam09] Peter J Cameron, « Oligomorphic permutation groups », in : *Perspectives in Mathematical Sciences II : Pure Mathematics*, World Scientific, 2009, p. 37-61.

[Cam83] Peter J Cameron, « Orbits of permutation groups on unordered sets, IV : homogeneity and transitivity », in : *Journal of the London Mathematical Society* 2.2 (1983), p. 238-247.

[Can+10] Ran Canetti *et al.*, « On symmetric encryption and point obfuscation », in : *Theory of Cryptography Conference*, Springer, 2010, p. 52-71.

[Can97] Ran Canetti, « Towards realizing random oracles : Hash functions that hide all partial information », in : *Annual International Cryptology Conference*, Springer, 1997, p. 455-469.

[Car11] Stephen R Carter, *Techniques to pollute electronic profiling*, US Patent 8,069,485, nov. 2011.

[CC02] Patrick Cousot et Radhia Cousot, « Modular static program analysis », in : *International Conference on Compiler Construction*, Springer, 2002, p. 159-179.

[CD05] Srinivasan Chandrasekharan et Saumya Debray, *Deobfuscation : Improving reverse engineering of obfuscated code*, rapp. tech., 2005.

[CD08] Ran Canetti et Ronny Ramzi Dakdouk, « Obfuscating point functions with multibit output », in : *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2008, p. 489-508.

[CDE+08] Cristian Cadar, Daniel Dunbar, Dawson R Engler *et al.*, « KLEE : Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. », in : *OSDI*, t. 8, 2008, p. 209-224.

[CE05] Cristian Cadar et Dawson Engler, « Execution generated test cases : How to make systems code crash itself », in : *International SPIN Workshop on Model Checking of Software*, Springer, 2005, p. 2-23.

[Cec+17] Mariano Ceccato *et al.*, « How professional hackers understand protected code while performing attack tasks », in : *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, IEEE, 2017, p. 154-164.

[Cec+19] Mariano Ceccato *et al.*, « Understanding the behaviour of hackers while performing attack tasks in a professional setting and in a public challenge », in : *Empirical Software Engineering* 24.1 (2019), p. 240-286.

[CFM12] Joan Calvet, José M Fernandez et Jean-Yves Marion, « Aligot : cryptographic function identification in obfuscated binary programs », in : *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, p. 169-182.

[CGM08] Peter J Cameron, Daniele A Gewurz et Francesca Merola, « Product action », in : *Discrete mathematics* 308.*2-3* (2008), p. 386-394.

[Cha+12] Sang Kil Cha *et al.*, « Unleashing mayhem on binary code », in : *2012 IEEE Symposium on Security and Privacy*, IEEE, 2012, p. 380-394.

[Cha+19] Sooyoung Cha *et al.*, « Enhancing Dynamic Symbolic Execution by Automatically Learning Search Heuristics », in : *arXiv preprint arXiv :1907.09700* (2019).

[Che+16] Jung Hee Cheon *et al.*, « Cryptanalysis of the new CLT multilinear map over the integers », in : *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2016, p. 509-536.

[Che+17] Ting Chen *et al.*, « Under-optimized smart contracts devour your money », in : *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*, IEEE, 2017, p. 442-446.

[Cho+01] Stanley Chow *et al.*, « An approach to the obfuscation of control-flow of sequential computer programs », in : *International Conference on Information Security*, Springer, 2001, p. 144-155.

[Cho+02] Stanley Chow *et al.*, « White-box cryptography and an AES implementation », in : *International Workshop on Selected Areas in Cryptography*, Springer, 2002, p. 250-270.

[CKC11] Vitaly Chipounov, Volodymyr Kuznetsov et George Candea, « S2E : A platform for in-vivo multi-path analysis of software systems », in : *ACM SIGARCH Computer Architecture News*, t. 39, 1, ACM, 2011, p. 265-278.

[CLD11] Kevin Coogan, Gen Lu et Saumya Debray, « Deobfuscation of virtualization-obfuscated software : a semantics-based approach », in : *Proceedings of the 18th ACM conference on Computer and communications security*, 2011, p. 275-284.

[CLT13] Jean-Sébastien Coron, Tancrede Lepoint et Mehdi Tibouchi, « Practical multilinear maps over the integers », in : *Annual Cryptology Conference*, Springer, 2013, p. 476-493.

[CMR17] Brent Carmer, Alex J Malozemoff et Mariana Raykova, « 5Gen-C : multi-input functional encryption and program obfuscation for arithmetic circuits », in : *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2017, p. 747-764.

[CMR98] Ran Canetti, Daniele Micciancio et Omer Reingold, « Perfectly one-way probabilistic hash functions (preliminary version) », in : *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, ACM, 1998, p. 131-140.

[Coh06] Fred Cohen, « The use of deception techniques : Honeypots and decoys », in : *Handbook of Information Security* 3.*1* (2006), p. 646-655.

[Coh93] Frederick B Cohen, « Operating system protection through program evolution. », in : *Computers & Security* 12.*6* (1993), p. 565-584.

[Col+12] Christian Collberg *et al.*, « Distributed application tamper detection via continuous software updates », in : *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012, p. 319-328.

[Col+19] Christian Collberg *et al.*, « The Tigress C diversifier/obfuscator », in : (2019), [Online ; accés 02-01-2020].

[Cor+16] Jean-Sébastien Coron *et al.*, « Cryptanalysis of GGH15 multilinear maps », in : *Annual International Cryptology Conference*, Springer, 2016, p. 607-628.

[CP10] Jan Cappaert et Bart Preneel, « A general model for hiding control flow », in : *Proceedings of the tenth annual ACM workshop on Digital rights management*, ACM, 2010, p. 35-42.

[CR91] Douglas Cenzer et Jeffrey Remmel, « Polynomial-time versus recursive models », in : *Annals of Pure and Applied Logic* 54.*1* (1991), p. 17-58.

[CRV10] Ran Canetti, Guy N Rothblum et Mayank Varia, « Obfuscation of hyperplane membership », in : *Theory of Cryptography Conference*, Springer, 2010, p. 72-89.

[CS18] Sylvain Cochon et Laurent Simon, « SatisfaisabilitéPropositionnelle (SAT) et-Modulo Théories (SMT) », in : 2018, chap. 2, url : `https://www.lri.fr/~conchon/FIIL/ejcim2018.pdf`.

[CTL97] Christian Collberg, Clark Thomborson et Douglas Low, *A taxonomy of obfuscating transformations*, rapp. tech., Department of Computer Science, The University of Auckland, New Zealand, 1997.

[CTL98] Christian Collberg, Clark Thomborson et Douglas Low, « Manufacturing cheap, resilient, and stealthy opaque constructs », in : *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM, 1998, p. 184-196.

[Dan+05] Sebastian Danicic *et al.*, « ConSUS : a light-weight program conditioner », in : *Journal of Systems and Software* 77.*3* (2005), p. 241-262.

[Dav+15] Lucas Davi *et al.*, « HAFIX : hardware-assisted flow integrity extension », in : ACM Press, 2015, p. 1-6, isbn : 978-1-4503-3520-1, doi : 10.1145/2744769.2744847, url : http://dl.acm.org/citation.cfm?doid=2744769.2744847 (visité le 20/07/2018).

[Dav+16] Robin David *et al.*, « BINSEC/SE : A dynamic symbolic execution toolkit for binary-level analysis », in : *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, t. 1, IEEE, 2016, p. 653-656.

[Dav17] Robin David, « Formal Approaches for Automatic Deobfuscation and Reverse-engineering of Protected Codes », thèse de doct., Université de Lorraine, 2017.

[DB08] Leonardo De Moura et Nikolaj Bjørner, « Z3 : An efficient SMT solver », in : *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2008, p. 337-340.

[DHK12] Vijay D'Silva, Leopold Haller et Daniel Kroening, « Satisfiability solvers are static analysers », in : *International Static Analysis Symposium*, Springer, 2012, p. 317-333.

[DKS14] Lucas Davi, Patrick Koeberl et Ahmad-Reza Sadeghi, « Hardware-assisted fine-grained control-flow integrity : Towards efficient protection of embedded systems against software exploitation », in : IEEE, juin 2014, p. 1-6, isbn : 978-1-4799-3017-3, doi : 10.1109/DAC.2014.6881460, url : http://ieeexplore.ieee.org/document/6881460/ (visité le 20/07/2018).

[DP08] Daniel Dolz et Gerardo Parra, « Using exception handling to build opaque predicates in intermediate code obfuscation techniques », in : *Journal of Computer Science & Technology* 8 (2008).

[Dul19] T. F. Dullien, « Weird machines, exploitability, and provable unexploitability », in : *IEEE Transactions on Emerging Topics in Computing* (2019), p. 1-1, issn : 2168-6750, doi : 10.1109/TETC.2017.2785299.

[Dun+12] Dmitriy Dunaev *et al.*, « Complexity of a special deobfuscation problem », in : *2012 IEEE 19th International Conference and Workshops on Engineering of Computer-Based Systems*, IEEE, 2012, p. 1-4.

[Elg61] Calvin C Elgot, « Decision problems of finite automata design and related arithmetics », in : *Transactions of the American Mathematical Society* 98.*1* (1961), p. 21-51.

[ES03] Niklas Eén et Niklas Sörensson, « An extensible SAT-solver », in : *International conference on theory and applications of satisfiability testing*, Springer, 2003, p. 502-518.

[Eyr17] Ninon Eyrolles, « Obfuscation with Mixed Boolean-Arithmetic Expressions : reconstruction, analysis and simplification tools », thèse de doct., Université Paris-Saclay, 2017.

[FS01] Mike Frantzen et Mike Shuey, « StackGhost : Hardware Facilitated Stack Protection », in : USENIX, USENIX, 2001.

[Gab14] F Gabriel, « Deobfuscation : recovering an OLLVM-protected program », in : *QuarkLabs* 4 (2014), p. 12.

[Gan+04] Harald Ganzinger *et al.*, « DPLL(T) : Fast Decision Procedures », in : *Computer Aided Verification*, sous la dir. de Rajeev Alur et Doron A. Peled, Berlin, Heidelberg : Springer Berlin Heidelberg, 2004, p. 175-188, isbn : 978-3-540-27813-9.

[Gar+13] Sanjam Garg *et al.*, « Candidate Indistinguishability Obfuscation and Functional Encryption for all Circuits », in : *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, IEEE, 2013, p. 40-49.

[Gar+16] Sanjam Garg *et al.*, « Hiding secrets in software : A cryptographic approach to program obfuscation », in : *Communications of the ACM* 59.*5* (2016), p. 113-120.

[GF19] Peter Garba et Matteo Favaro, « SATURN-Software Deobfuscation Framework Based On LLVM », in : *Proceedings of the 3rd ACM Workshop on Software Protection*, 2019, p. 27-38.

[GGH15] Craig Gentry, Sergey Gorbunov et Shai Halevi, « Graph-induced multilinear maps from lattices », in : *Theory of Cryptography Conference*, Springer, 2015, p. 498-527.

[GK05] Shafi Goldwasser et Yael Tauman Kalai, « On the impossibility of obfuscation with auxiliary input », in : *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)*, IEEE, 2005, p. 553-562.

[GK13] Shafi Goldwasser et Yael Tauman Kalai, « A Note on the Impossibility of Obfuscation with Auxiliary Input. », in : *IACR Cryptology ePrint Archive* 2013 (2013), p. 665.

[GL19] Alexandre Gonzalvez et Ronan Lashermes, « A case against indirect jumps for secure programs », in : *SSPREW9 '19*, 2019.

[GLM12] Patrice Godefroid, Michael Y Levin et David Molnar, « SAGE : whitebox fuzzing for security testing », in : *Queue* 10.*1* (2012), p. 20-27.

[God12] Patrice Godefroid, « Test generation using symbolic execution », in : *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2012)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2012.

[Gol11] Oded Goldreich, « Basic facts about expander graphs », in : *Studies in Complexity and Cryptography. Miscellanea on the Interplay between Randomness and Computation*, Springer, 2011, p. 451-464.

[GR14] Shafi Goldwasser et Guy N. Rothblum, « On Best-Possible Obfuscation », in : *Journal of Cryptology* 27.*3* (juil. 2014), p. 480-505, issn : 0933-2790, 1432-1378, doi : 10.1007/s00145-013-9151-z.

[HB15] Máté Horváth et Levente Buttyán, *The birth of cryptographic obfuscation-A survey*, rapp. tech., Cryptology ePrint Archive, Report 2015/412, 2015.

[HD11] James Hamilton et Sebastian Danicic, « A survey of static software watermarking », in : *Internet Security (WorldCIS), 2011 World Congress on*, IEEE, 2011, p. 100-107.

[HJ16] Yupu Hu et Huiwen Jia, « Cryptanalysis of GGH map », in : *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2016, p. 537-565.

[Hod82] Bernard R Hodgson, « On direct products of automaton decidable theories », in : *Theoretical Computer Science* 19.*3* (1982), p. 331-335.

[Hos+18] Shohreh Hosseinzadeh *et al.*, « Diversification and obfuscation techniques for software security : A systematic literature review », in : *Information and Software Technology* (2018).

[HSG17] T. Hiscock, O. Savry et L. Goubin, « Lightweight Software Encryption for Embedded Processors », in : *2017 Euromicro Conference on Digital System Design (DSD)*, août 2017, p. 213-220, doi : 10.1109/DSD.2017.25.

[HW+93] Wilfrid Hodges, Hodges Wilfrid *et al.*, *Model theory*, t. 42, Cambridge University Press, 1993.

[IA07] AA Ibrahim et MS Audu, « On wreath Product of permutation Groups », in : *Proyecciones (Antofagasta)* 26.*1* (2007), p. 73-90.

[Imm12] Neil Immerman, *Descriptive complexity*, Springer Science & Business Media, 2012.

[Jun+15] Pascal Junod *et al.*, « Obfuscator-LLVM–software protection for the masses », in : *2015 IEEE/ACM 1st International Workshop on Software Protection*, IEEE, 2015, p. 3-9.

[Kaf07] Ben Kafka, « The demon of writing : paperwork, public safety, and the reign of terror », in : *Representations* 98.*1* (2007), p. 1-24.

[KG07] Sava Krstić et Amit Goel, « Architecting solvers for SAT modulo theories : Nelson-Oppen with DPLL », in : *International Symposium on Frontiers of Combining Systems*, Springer, 2007, p. 1-27.

[Kho+04] Bakhadyr Khoussainov *et al.*, « Automatic structures : richness and limitations », in : *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004.* IEEE, 2004, p. 44-53.

[Kin75] James C. King, « A New Approach to Program Testing », in : *SIGPLAN Not.* 10.*6* (avr. 1975), p. 228-233, issn : 0362-1340, doi : 10.1145/390016.808444, url : http://doi.acm.org/10.1145/390016.808444.

[KK12] Johannes Kinder et Dmitry Kravchenko, « Alternating control flow reconstruction », in : *International Workshop on Verification, Model Checking, and Abstract Interpretation*, Springer, 2012, p. 267-282.

[KL80] Richard M Karp et Richard J Lipton, « Some connections between nonuniform and uniform complexity classes », in : *Proceedings of the twelfth annual ACM symposium on Theory of computing*, ACM, 1980, p. 302-309.

[KM07] Bakhadyr Khoussainov et Mia Minnes, « Three lectures on automatic structures », in : *Proceedings of Logic Colloquium*, 2007, p. 132-176.

[KN94] Bakhadyr Khoussainov et Anil Nerode, « Automatic presentations of structures », in : *International Workshop on Logic and Computational Complexity*, Springer, 1994, p. 367-392.

[Kop11] Vera Koponen, « Some connections between finite and infinite model theory », in : (2011).

[KS16] Daniel Kroening et Ofer Strichman, *Decision procedures*, Springer, 2016.

[KW11] Dietrich Kuske et Thomas Weidner, « Size and computation of injective tree automatic presentations », in : *International Symposium on Mathematical Foundations of Computer Science*, Springer, 2011, p. 424-435.

[KZV09] Johannes Kinder, Florian Zuleger et Helmut Veith, « An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries », in : *Verification, Model Checking, and Abstract Interpretation*, sous la dir. de Neil D. Jones et Markus Müller-Olm, Berlin, Heidelberg : Springer Berlin Heidelberg, 2009, p. 214-228, isbn : 978-3-540-93900-9.

[Lap18] Julien Vanegue Laphroaig Manul, *PoC or GTFO*, t. 0x18, 2018, p. 51-57.

[Lat08] Chris Lattner, « LLVM and Clang : Next generation compiler technology », in : *The BSD conference*, t. 5, 2008.

[LB84] Hector J Levesque et Ronald J Brachman, *A fundamental tradeoff in knowledge representation and reasoning*, Laboratory for Artificial Intelligence Research, Fairchild, Schlumberger, 1984.

[LD03] Cullen Linn et Saumya Debray, « Obfuscation of executable code to improve resistance to static disassembly », in : *Proceedings of the 10th ACM conference on Computer and communications security*, ACM, 2003, p. 290-299.

[Lia+17] Mingyue Liang *et al.*, « Deobfuscation of virtualization-obfuscated code through symbolic execution and compilation optimization », in : *International Conference on Information and Communications Security*, Springer, 2017, p. 313-324.

[Lin17] Yehuda Lindell, « How to simulate it–a tutorial on the simulation proof technique », in : *Tutorials on the Foundations of Cryptography*, Springer, 2017, p. 277-346.

[Luo+14] Lannan Luo *et al.*, « Semantics-Based Obfuscation-Resilient Binary Code Similarity Comparison with Applications to Software Plagiarism Detection », in : *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, Hong Kong, China : Association for Computing Machinery, 2014, p. 389-400, ISBN : 9781450330565, DOI : 10.1145/2635868.2635900, URL : https://doi.org/10.1145/2635868.2635900.

[Mac11] Dugald Macpherson, « A survey of homogeneous structures », in : *Discrete Mathematics* 311.*15* (2011), p. 1599-1634.

[Man+16] Ramya Manikyam *et al.*, « Comparing the effectiveness of commercial obfuscators against MATE attacks », in : *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering*, ACM, 2016, p. 8.

[McD12] J Todd McDonald, « Capturing the essence of practical obfuscation », in : *International Conference on Information Systems, Technology and Management*, Springer, 2012, p. 451-456.

[Min+15] Jiang Ming *et al.*, « Loop : Logic-oriented opaque predicate detection in obfuscated binary code », in : *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2015, p. 757-768.

[Mor76] Michael Morley, « Decidable models », in : *Israel Journal of Mathematics* 25.*3-4* (1976), p. 233-240.

[Mos+01] Matthew W Moskewicz *et al.*, « Chaff : Engineering an efficient SAT solver », in : *Proceedings of the 38th annual Design Automation Conference*, 2001, p. 530-535.

[MP88] Farhad Mavaddat et Behrooz Parhami, « URISC : the ultimate reduced instruction set computer », in : *International Journal of Electrical Engineering Education* 25.*4* (1988), p. 327-334.

[MS05] Orly Meir et Ofer Strichman, « Yet another decision procedure for equality logic », in : *International Conference on Computer Aided Verification*, Springer, 2005, p. 307-320.

[MT06] Anirban Majumdar et Clark Thomborson, « Manufacturing opaque predicates in distributed systems for code obfuscation », in : *Proceedings of the 29th Australasian Computer Science Conference-Volume 48*, Australian Computer Society, Inc., 2006, p. 187-196.

[MVD06] Matias Madou, Ludo Van Put et Koen De Bosschere, « Loco : An interactive code (de) obfuscation tool », in : *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, 2006, p. 140-144.

[NC09] Jasvir Nagra et Christian Collberg, *Surreptitious software : obfuscation, watermarking, and tamperproofing for software protection*, Pearson Education, 2009.

[Nej20] Saeed Nejati, « CDCL (Crypto) and Machine Learning based SAT Solvers for Cryptanalysis », in : (2020).

[Nie+18] Aina Niemetz *et al.*, « Btor2, btormc and boolector 3.0 », in : *International Conference on Computer Aided Verification*, Springer, 2018, p. 587-595.

[NPB14] Aina Niemetz, Mathias Preiner et Armin Biere, « Boolector 2.0 », in : *Journal on Satisfiability, Boolean Modeling and Computation* 9.*1* (2014), p. 53-58.

[NPB19] Aina Niemetz, Mathias Preiner et Armin Biere, « Boolector at the SMT competition 2019 », in : *Proceedings of the 17th International Workshop on Satisfiability Modulo Theories (SMT 2019), affiliated with the 22nd International Conference on Theory and Applications of Satisfiability Testing (SAT 2019), Lisbon, Portugal, July 7-8, 2019*, sous la dir. de Joe Hendrix et Natasha Sharygina, 2019, 2 pages.

[Oll+19a] Mathilde Ollivier *et al.*, « How to Kill Symbolic Deobfuscation for Free ; or Unleashing the Potential of Path-Oriented Protections », in : *arXiv preprint arXiv :1908.01549* (2019).

[Oll+19b] Mathilde Ollivier *et al.*, « Obfuscation : where are we in anti-DSE protections ?(a first attempt) », in : *Proceedings of the 9th Workshop on Software Security, Protection, and Reverse Engineering*, 2019, p. 1-8.

[OT05] Graham P Oliver et Richard M Thomas, « Automatic presentations for finitely generated groups », in : *Annual Symposium on Theoretical Aspects of Computer Science*, Springer, 2005, p. 693-704.

[Pal+00] Jens Palsberg *et al.*, « Experience with software watermarking », in : *Computer Security Applications, 2000. ACSAC'00. 16th Annual Conference*, IEEE, 2000, p. 308-316.

[Par13] Arash Partow, *General purpose hash function algorithms*, `https://www.partow.net/programming/hashfunctions/`, Accessed : 2019-09-01, 2013.

[PD10] Knot Pipatsrisawat et Adnan Darwiche, « On modern clause-learning satisfiability solvers », in : *Journal of Automated Reasoning* 44.*3* (2010), p. 277-301.

[Rin17] Thomas Rinsma, « Seeing through obfuscation : interactive detection and removal of opaque predicates », thèse de doct., Radboud University, 2017.

[RKG18] Robert Robere, Antonina Kolokolova et Vijay Ganesh, « The Proof Complexity of SMT Solvers », in : *International Conference on Computer Aided Verification*, Springer, 2018, p. 275-293.

[Roe+12] Ryan Roemer *et al.*, « Return-Oriented Programming : Systems, Languages, and Applications », in : *ACM Trans. Inf. Syst. Secur.* 15.*1* (mar. 2012), 2 :1-2 :34, issn : 1094-9224, doi : 10.1145/2133375.2133377, url : http://doi.acm.org/10.1145/2133375.2133377.

[Rom81] N S Romanovskiĭ, « ON THE ELEMENTARY THEORY OF AN ALMOST POLYCYCLIC GROUP », in : *Mathematics of the USSR-Sbornik* 39.*1* (fév. 1981), p. 125-132, doi : 10.1070/sm1981v039n01abeh001476, url : https://doi.org/10.1070%2Fsm1981v039n01abeh001476.

[Rua+16] Ruan de Clercq *et al.*, « SOFIA : Software and Control Flow Integrity Architecture », in : DATE, IEEE, 2016.

[SA15] Jia Song et Jim Alves-Foss, « The darpa cyber grand challenge : A competitor's perspective », in : *IEEE Security & Privacy* 13.*6* (2015), p. 72-76.

[SBP18] Jonathan Salwan, Sébastien Bardin et Marie-Laure Potet, « Symbolic deobfuscation : From virtualized code back to the original », in : *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2018, p. 372-392.

[Sch+16] Sebastian Schrittwieser *et al.*, « Protecting software through obfuscation : Can it keep pace with progress in code analysis ? », in : *ACM Computing Surveys (CSUR)* 49.*1* (2016), p. 4.

[SH12] Michael Sikorski et Andrew Honig, *Practical malware analysis : the hands-on guide to dissecting malicious software*, no starch press, 2012.

[Sha+08] Monirul I Sharif *et al.*, « Impeding Malware Analysis Using Conditional Code Obfuscation. », in : *NDSS*, 2008.

[Sho+16] Yan Shoshitaishvili *et al.*, « Sok :(state of) the art of war : Offensive techniques in binary analysis », in : *2016 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2016, p. 138-157.

[SI14] Vladimir Sergeichik et Alexander Ivaniuk, « Implementation of opaque predicates for FPGA designs hardware obfuscation », in : (2014).

[SS15] Jonathan Salwan et Florent Saudel, « Triton : Framework d'exécution concolique et d'analyses en runtime », in : *Proceedings of the 2015 Symposium sur la Securite des Technologies de l'Information et des Communications, ser. SSTIC*, t. 15, 2015.

[SW14] Amit Sahai et Brent Waters, « How to use indistinguishability obfuscation : deniable encryption, and more », in : *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, 2014, p. 475-484.

[Tar72] Alfred Tarski, « Logique Sémantique, Métamathématique, 1923-1944 », in : (1972).

[TH20] Yu-Jye Tung et Ian G. Harris, « A Heuristic Approach to Detect Opaque Predicatesthat Disrupt Static Disassembly », in : *NDSS*, 2020.

[The00] H. Theiling, « Extracting safe and precise control flow from binaries », in : *Proceedings Seventh International Conference on Real-Time Computing Systems and Applications*, déc. 2000, p. 23-30, doi : 10.1109/RTCSA.2000.896367.

[Tic+] Caroline Tice *et al.*, « Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM », in : (), p. 15.

[Tin10] Cesare Tinelli, « Foundations of Satisfiability Modulo Theories. », in : *WoLLIC*, 2010, p. 58.

[Tof+19] Ramtine Tofighi-Shirazi *et al.*, « Defeating Opaque Predicates Statically through Machine Learning and Binary Analysis », in : *Proceedings of the 3rd ACM Workshop on Software Protection*, 2019, p. 3-14.

[UDM05] Sharath K Udupa, Saumya K Debray et Matias Madou, « Deobfuscation : Reverse engineering obfuscated code », in : *12th Working Conference on Reverse Engineering (WCRE'05)*, IEEE, 2005, 10-pp.

[Wan+00] Chenxi Wang *et al.*, *Software tamper resistance : Obstructing static analysis of programs*, rapp. tech., Technical Report CS-2000-12, University of Virginia, 12 2000, 2000.

[Wan+11] Zhi Wang *et al.*, « Linear obfuscation to combat symbolic execution », in : *European Symposium on Research in Computer Security*, Springer, 2011, p. 210-226.

[Wee05] Hoeteck Wee, « On obfuscating point functions », in : *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, ACM, 2005, p. 523-532.

[Wer97] Benjamin Werner, « Sets in types, types in sets », in : *International Symposium on Theoretical Aspects of Computer Software*, Springer, 1997, p. 530-546.

[WG00] David Wagner et Ian Goldberg, « Proofs of security for the Unix password hashing algorithm », in : *International Conference on the Theory and Application of Cryptology and Information Security*, Springer, 2000, p. 560-572.

[Woo+14] Jonathan Woodruff *et al.*, « The CHERI capability model : Revisiting RISC in an age of risk », in : *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, IEEE, 2014, p. 457-468.

[Xu+17] Hui Xu *et al.*, « On Secure and Usable Program Obfuscation : A Survey », in : *arXiv preprint arXiv :1710.01139* (2017).

[Xu+18] Hui Xu *et al.*, « Manufacturing Resilient Bi-Opaque Predicates Against Symbolic Execution », in : *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, IEEE, 2018.

[Yad+15] Babak Yadegari *et al.*, « A generic approach to automatic deobfuscation of executable code », in : *2015 IEEE Symposium on Security and Privacy*, IEEE, 2015, p. 674-691.

[Yad16] Babak Yadegari, « Automatic deobfuscation and reverse engineering of obfuscated code », in : (2016).

[YY10] Ilsun You et Kangbin Yim, « Malware obfuscation techniques : A brief survey », in : *2010 International conference on broadband, wireless computing, communication and applications*, IEEE, 2010, p. 297-300.

[ZGR17] Lukas Zobernig, Steven D. Galbraith et Giovanni Russello, *When Are Opaque Predicates Useful ?*, Cryptology ePrint Archive, Report 2017/787, `https://eprint.iacr.org/2017/787`, 2017.

Diffusions des résultats

Workshops

- Software Security, Protection, and Reverse Engineering Workshop (SSPREW), San Juan, Porto Rico, USA, 9-10 Décembre, 2019 ;
- SecDays Workshop, Rennes, France, Janvier 2019 ;

Présentations

- Équipe CIDRE (*Centrale-Supélec, campus de Rennes*)
- Équipe TAMIS (*Inria-RBA*)
- Journée « Protection du code et des données, obfuscation & whitebox cryptography », Paris-Saclay, France, 13 décembre 2018

Autres

- Requêtes pour la compétition SMT-COMP 2019

Poster

La page suivante contient une reproduction du poster présenté lors des SecDays à Inria-RBA en Janvier 2019 à Rennes.

Opacity properties and SMT solvers

Alexandre Gonzalvez*[1,2], Olivier Decourbe[2], Sebastien Josse[3],
Caroline Fontaine[4], Axel Legay[5]

1 IMT atlantique, CNRS, Lab-STICC, France
2 Inria, Univ Rennes, CNRS, IRISA, France
3 DGA Maitrise de l'information, France
4 LSV, CNRS & ENS-Saclay, Université Paris-Saclay, France
5 UCLouvain, Belgium
* contact: alexandre.gonzalvez@inria.fr

Abstract

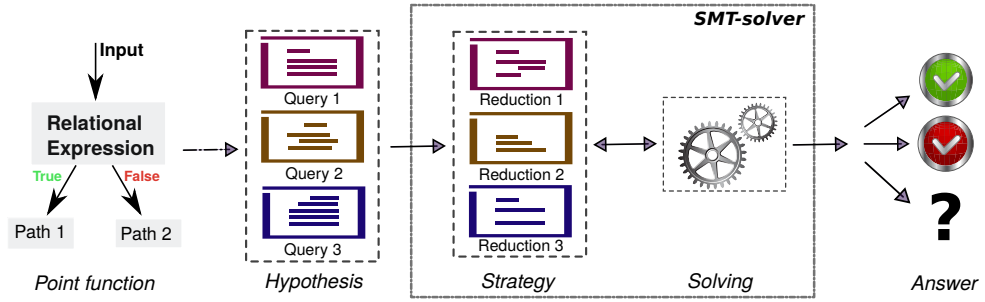
Reverse engineering can be automated with some binary or source code analysis frameworks and techniques. They can be used, for example, to detect potential vulnerabilities or to detect malware, by applying the field of mathematical logic.

1 This abstraction can be realized with a framework and sent to an SMT (Satisfiability Modulo Theories) solver, which checks satisfiability of given hypothesis in regards to some background theory, and approximations.

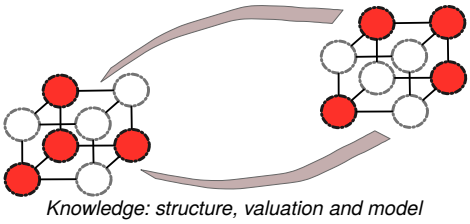
2 This work proposes an attacker point of view against opacity properties, i.e. why opacity properties have a negative impact in the learning process made by an SMT solver.

3 Experience shows that hypothesis need to be rewritten and adapted for each opacity property in the aim to reduce the time for the analysis i.e. the number of steps to learn a concept composed with these properties.

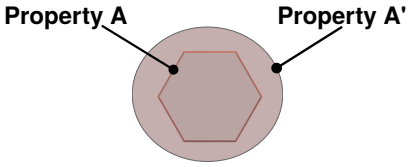
1



2

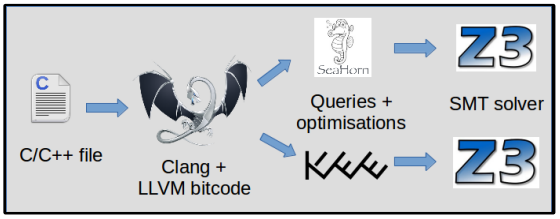


Knowledge: structure, valuation and model



Hypothesis and Approximation

3



Simplified architecture of Seahorn (Gurfinkel et al.), and Simplified architecture of KLEE (Cadar et al.)

Input size	KLEE-Z3	Seahorn-Z3	Seahorn + Rewriting
5 char	952 s.	UNSAT	47 s.
10 char	184 s.	UNSAT	55 s.
15 char	TO	UNSAT	80 s.

TO : Time Out (20 min)

Results of our experience (time analysis) with the APartow hash function composed with a constant expression with free variables: $x * (x+1)\%2$

Iconography credits: http:\\www.all-free-dowload.com

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

AVIS DE SOUTENANCE DE THESE

DOCTORAT BRETAGNE LOIRE MATHSTIC

M. Alexandre GONZALVEZ

Dpt ITI – Laboratoire Lab-STICC

Soutiendra publiquement ses travaux en vue de l'obtention du grade de

Docteur d'IMT Atlantique

Dans le cadre de la co-accréditation de thèse d'IMT Atlantique

Le 02/06/2020 à 13:00 en visio-conférence

(dispositions exceptionnelles durant la crise sanitaire liée au Covid19).

Affiner la déobfuscation symbolique et concrète de programmes protégés par des prédicats opaques

Résumé : Une forte demande existe aujourd'hui pour améliorer les techniques avancées d'obfuscation et de déobfuscation, dans le but d'éviter le vol de propriétés intellectuelles ou de perfectionner la défense face aux attaques en cybersécurité.

Les travaux réalisés au cours de cette thèse portent sur la consolidation de la déobfuscation réalisée par des outils d'analyse symbolique et concrète de programmes protégés par des prédicats opaques. Ces outils s'appuient sur des outils d'analyse automatisée de programmes (moteur d'exécution symbolique dynamique) qui utilisent des solveurs de satisfiabilité modulo théorie (solveurs SMT). Nous souhaitons comprendre plus précisément certaines situations pour lesquelles l'analyse de prédicats effectuée par ces outils est mise en échec, pour ensuite pouvoir proposer des solutions pratiques évitant ces scénarios, et les tester dans des cas réels. C'est pourquoi notre travail se concentre sur la compréhension des concepts de raisonnement automatisé dans une théorie décidable et complète, afin de clarifier les mécanismes de capture de l'information.

Nos premiers résultats montrent comment un jeu d'instructions machine assembleur (ISA) autorise l'apparition ou non de prédicats opaques. Nous proposons une amélioration de la détection de prédicats opaques à partir du comportement du solveur SMT. Nous proposons une redéfinition des requêtes SMT pour réduire les effets des prédicats opaques. Nous intégrons ces améliorations dans plusieurs outils automatiques tels que KLEE ou Angr, puis les testons sur différents programmes contenant des prédicats opaques.

Mots-clés: Obfuscation ; Moteur d'exécution symbolique dynamique ; Solveur SMT ; Prédicats Opaques

Le jury est composé de :

- M. Sébastien BARDIN	Docteur, Ingénieur	CEA
- M. Fabien DAGNAT	Maître de conférences	IMT Atlantique
- Mme Caroline FONTAINE	Directeur de recherche	LSV, CNRS & ENS Paris-Saclay
- M. Jean-Louis LANET	Professeur	INRIA
- M. Sébastien JOSSE	Docteur, Ingénieur	DGA
- Mme Marion VIDEAU	Responsable Scientifique	QUARKSLAB
- M. Pascal LAFOURCADE	HDR	Université Clermont Auvergne
- M. Daniel LE BERRE	Professeur	Université d'Artois

UNIVERSITE BRETAGNE LOIRE MATHSTIC

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Titre : Affiner la déobfuscation symbolique et concrète de programmes protégés par des prédicats opaques

Mot clés : Obfuscation, Moteur d'exécution symbolique dynamique, Solveur SMT, Prédicats Opaques

Résumé : Une forte demande existe aujourd'hui pour améliorer les techniques avancées d'obfuscation et de déobfuscation, dans le but d'éviter le vol de propriétés intellectuelles ou de perfectionner la défense face aux attaques en cybersécurité. Les travaux réalisés au cours de cette thèse portent sur la consolidation de la déobfuscation réalisée par des outils d'analyse symbolique et concrète de programmes protégés par des *prédicats opaques*. Ces outils s'appuient sur des outils d'analyse automatisée de programmes (*moteur d'exécution symbolique dynamique*) qui utilisent des solveurs de *satisfiabilité modulo théorie* (*solveurs SMT*). Nous souhaitons comprendre plus précisément certaines situations pour lesquelles l'analyse de prédicats effectuée par ces outils est mise en échec, pour ensuite pouvoir proposer des solutions pratiques évitant ces scénarios, et les tester dans des cas réels. C'est pourquoi notre travail se concentre sur la compréhension des concepts de raisonnement automatisé dans une théorie décidable et complète, afin de clarifier les mécanismes de capture de l'information. Nos premiers résultats montrent comment un *jeu d'instructions machine assembleur* (*ISA*) autorise l'apparition ou non de *prédicats opaques*. Nous proposons une amélioration de la détection de prédicats opaques à partir du comportement du solveur SMT. Nous proposons une redéfinition des requêtes SMT pour réduire les effets des prédicats opaques. Nous intégrons ces améliorations dans plusieurs outils automatiques tels que *KLEE* ou *Angr*, puis les testons sur différents programmes contenant des *prédicats opaques*.

Title: To refine symbolic and concrete deobfuscation of programs protected by opaque predicates

Keywords: Obfuscation, Dynamic Symbolic Execution Engine, SMT Solver, Opaque Predicates

Abstract: High demand exists nowadays to improve advanced obfuscation and deobfuscation techniques, with the purpose of preventing intellectual property piracy or improving defence against cyber security attacks. This thesis focuses on the improvement of the deobfuscation achieved by symbolic and concrete analysis tools of protected programs using *opaque predicates*. These tools rely on automated program analysis tools (*dynamic symbolic execution engine*) that use Satisfiability Modulo Theory solvers (*SMT solvers*). To understand more precisely some situations in which the predicate analysis performed by these tools fails, our aim is to be able to identify practical solutions to avoid these scenarios and test them in real cases. First results show how an Instruction Set Assembly (*ISA*) allows *opaque predicates* to appear or not. We suggest an improvement of the *opaque predicates* identification based on the *SMT solvers* behavior. We suggest a method to reshape SMT queries to reduce the effects of opaque predicates. These features are built into several automated tools such as *KLEE* or *Angr*, followed by testing them on different programs which contain *opaque predicates*.