L'usage de l'exécution symbolique pour la déobfuscation binaire en milieu industriel

Jonathan Salwan

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITE GRENOBLE ALPES

Spécialité : Mathématiques et Informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Jonathan SALWAN

Thèse dirigée par **Marie laure POTET,** Grenoble INP et codirigée par **Sébastien BARDIN,** CEA LIST préparée au sein du laboratoire **Quarkslab,** du laboratoire **VERIMAG** et du laboratoire **CEA LIST**.

dans l'**École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

L'usage de l'exécution symbolique pour la déobfuscation binaire en milieu industriel

Thèse soutenue publiquement le **13 février 2020 à Grenoble**, devant le jury composé de :

Pr. Vincent Nicomette
Professeur INSA Toulouse LAAS-CNRS, Rapporteur
Pr. Jean-Yves Marion
Professeur Université de Lorraine, Rapporteur
Pr. Philippe Elbaz-Vincent
Professeur Université Grenoble Alpes, Président de jury
Dr. Sarah Zennou
Docteur Ingénieur Recherche Airbus, Examinateur
Dr. Robin David
Docteur Ingénieur Recherche Quarkslab, Examinateur
Pr. Marie-Laure Potet
Professeur Grenoble INP, Directeur de thèse
Dr. Sébastien Bardin
Docteur Ingénieur recherche CEA, Co-encadrant
Cédric Tessier
Ingénieur Sqreen, Invité

Sois toujours comme la mer qui,
se brisant contre les rochers,
trouve toujours la force de
recommencer.

Jim Morrison

Remerciements

J'aimerais tout particulièrement remercier Fred Raynal pour sa confiance depuis toutes ces années ainsi que la liberté qu'il m'accorde dans les choix de sujets et les lieux de travail. J'aimerais également remercier Quarkslab pour le financement de cette thèse sans qui elle n'aurait jamais vu le jour. Je ne saurais suffisamment remercier mes encadrants Marie-Laure Potet, Sébastien Bardin, Marion Videau et Cédric Tessier pour leur patience, leur générosité à l'égard du partage de leurs connaissances, et surtout, pour avoir su me *reverser*. Merci à Philippe Elbaz-Vincent, Vincent Nicomette, Jean-Yvers Marion, Sarah Zennou, Robin David et Cédric Tessier d'avoir accepté de faire partie de mon jury. Un grand merci à tous les relecteurs spontanés Serge Guelton, Camille Mougey, Guillaume Heilles, Robin David et Stéphane Miège qui ont su rajouter leur grain de sel à cette thèse. J'aimerais également remercier toutes les personnes avec qui j'ai pu discuter concernant les challenges de la rétro-ingénierie et qui ont grandement contribué à cette thèse : Stanislas Lejay, Philippe Teuwen, Guillaume Heilles, Adrien Guinet, Iván Arce, Benoit Lemarchand, Matthieu Daumas, Gabrielle Viala, Axel Souchet, Jérémy Fetiveau, Thomas Dullien, Stégan le Berre, Nicolas Correia et tous ceux qui veulent rester anonymes. Pour le cadre de vie qu'elle nous a offert, je tiens à dire un énorme merci à Isabelle Devaux sans qui cette thèse n'aurait sans doute pas pu aboutir. Merci à *VA* pour toutes ces heures loin de tout et où seul l'instant présent importe. Présent dont le temps semble s'être arrêté un instant. Solène Devaux, pour ton sourire, ta gentillesse, ta bonne humeur, ton amour et tes gâteaux ! Enfin, j'aimerais remercier mes parents et ma petite sœur qui ont toujours été là pour me soutenir peu importe mes décisions, même les plus folles (2021 la tête dans les étoiles...) et pour qui mon amour est sans limite.

Résumé

Cette thèse a été faite dans un cadre industriel où les activités principales sont la rétro-ingénierie pour la recherche de vulnérabilités et la vérification de certaines propriétés de sécurité sur des programmes déjà compilés. La première partie de cette thèse porte sur la collecte et le partage des problématiques industrielles lors de l'analyse de programmes binaires. Basé sur ces problématiques, un cadre de travail d'analyse dynamique binaire a été développé et formalisé. Des exemples réels d'utilisation y sont ensuite présentés tels que la détection de prédicats opaques dans les conditions de branchement. Enfin, une nouvelle approche automatique permettant la dévirtualisation de code binaire y est présentée en combinant les fonctionnalités du cadre de travail ainsi que celles d'autres outils.

Mots-clés : Rétro-ingénierie, Analyse binaire dynamique, Déobfuscation binaire, Exécution symbolique.

Abstract

This doctoral work has been done in an industrial environment where the main activities were reverse engineering for vulnerability research and security properties verification on binary programs. The first part of this doctoral work focuses on the collection and sharing of the industrial problems when analyzing binary programs. Based on these issues, a binary dynamic analysis framework has been developed and formalized. Real examples of use are then presented, such as the detection of opaque predicates in branch conditions. Finally, a new automatic approach for deobfuscation of binary code protected by virtualization is presented combining features of the framework as well as those of other tools.

Keywords : Reverse-engineering, Dynamic binary analysis, Deobfuscation, Symbolic execution.

Table des matières

Chapitre 1

Introduction

1.1 Le contexte

La sûreté et la sécurité des systèmes informatiques est de nos jours un point crucial et est au centre de toutes les préoccupations. La vérification logicielle a pour but de déterminer si un programme répond bien à des propriétés attendues que ce soit en termes de sécurité ou fonctionnellement. Un système étant composé d'applications, celles-ci doivent donc êtres soumises à des vérifications pour éviter tout comportement inattendu, on parle alors de mission d'audit (ou d'évaluation) de sécurité. Il est possible d'effectuer ces vérifications depuis le code source du programme ainsi que sur sa version compilée. Dans cette thèse nous considérons l'évaluation de programmes compilés [1].

Cette thèse est issue d'un retour d'expérience du monde de l'industrie dans le domaine de la sécurité informatique des applications compilées. Elle vise à mettre en évidence les contraintes et les problématiques industrielles (en termes de difficultés, de temps, de résultats, de cibles de travail, d'outils à disposition, etc.) pour l'analyse de programmes compilés, et à proposer des solutions académiques à la résolution de certaines de ces problématiques industrielles.

Cette thèse est financée par la société Quarkslab. Cette entreprise possède une forte expérience dans l'analyse de programmes compilés pour la recherche de vulnérabilités ou la vérification de propriétés (rétro-ingénierie, comportements

1. L'analyse de programmes compilés représente la majorité des missions d'audit de sécurité au sein de Quarkslab.

conformes, etc.). Cette thèse est co-encadrée par le laboratoire VERIMAG ainsi que le CEA LIST, qui tous deux possèdent de grandes compétences dans l'analyse formelle de programmes. L'objectif de cette thèse est de faire le lien entre les problématiques industrielles et les solutions (outils, méthodologies, cadres théoriques) que peut apporter le monde académique.

1.2 Contexte : mission d'audit de sécurité

Prenons l'exemple d'une société X voulant mettre sur le marché une application traitant les données confidentielles de ses utilisateurs. L'application, reflétant l'image de la société X, est généralement soumise à un audit de sécurité afin de limiter au maximum les vulnérabilités qui peuvent s'y trouver. Si de grosses vulnérabilités venaient à être découvertes après la mise en service de l'application, cela pourrait fortement affecter l'image de la société ainsi que celle de l'application.

Pour éviter de telles conséquences, la société X peut faire appel à des prestataires afin d'auditer son produit. Généralement, la société X effectue un appel d'offre mettant en avant ses besoins. Les prestataires, quant à eux, répondent à l'appel d'offre en mettant en avant leurs compétences, expériences et outillages afin de se *valoriser* auprès de la société X. Une fois le prestataire choisi, commence la mission dans un temps fixé par le client et le prestataire en fonction des besoins de la société X.

1.2.1 Méthodologie habituelle d'analyse

Chaque analyste possède sa propre méthodologie de travail en fonction des cibles qu'il audite. Toutefois, on peut constater qu'en moyenne les analystes suivent une méthodologie commune. Ils commencent par *jouer* avec l'application pour voir les fonctionnalités qu'elle propose. Ensuite, ils regardent les documentations techniques et utilisateurs (publics ou données par le client) afin d'avoir une vue globale de ce qu'il est possible de faire avec l'application. Vient ensuite le moment où l'on commence à regarder le code (source ou binaire) afin de transposer les fonctionnalités que l'on a identifiées précédemment avec le code que l'on a sous les yeux. L'identification visuelle d'une fonctionnalité permet de mieux comprendre le code qui la traite. À partir de là, l'analyse peut commencer à réfléchir à un moyen de détourner les fonctionnalités initiales de l'application par le biais de vulnérabilités logiques (ex. mauvaise conception) ou d'implémentation (ex. *buffer*

overflow) qu'il aura identifiées.

1.2.2 Méthodes d'analyse et importance de l'outillage

L'outillage, qu'il soit personnel ou public, fait partie intégrante du processus d'analyse. Les outils permettent de rapidement extraire des informations et laissent aux analystes la liberté de les interpréter. La connaissance et l'usage de bons outils permettent de gagner un temps précieux sur des missions dont le temps est limité. Par exemple, l'analyste peut être amené à utiliser une analyse statique sur un code binaire avec comme vue le graphe de flot de contrôle (*CFG*) du programme ou encore son flot de données (*DFG*). En reposant sur une vue statique, il peut également utiliser des outils d'analyse dynamique tels que des débogueurs afin de connaître certaines valeurs au moment de l'exécution. La majorité des analystes utilisent uniquement un désassembleur, un débogueur ainsi que leur expérience pour analyser un programme. Cependant il existe une grande variété d'outils que les analystes peuvent utiliser (liste non exhaustive) :

— **Outils d'analyse statique** : Outils permettant d'extraire des informations sur l'ensemble d'un programme sans l'exécuter tels que les outils de désassemblage ([77, 54, 103, 74, 7]), de signature ([53, 8]), de traitement de formats ([91, 52]), de décompilation ([54, 10]), etc.

— **Outils d'analyse dynamique** : Outils permettant d'extraire des informations au moment de l'exécution d'un programme tels que les outils de débogage ([89, 79]), de traçage ([78, 24, 72]), etc.

— **Outils de cloisonnement** : Outils permettant de cloisonner l'exécution d'un programme dans un environnement contrôlé tels que des virtualiseurs ([21, 81]), hyperviseurs ([95]), etc.

1.3 Les défis qui ont guidé nos travaux

Les défis lors d'une rétro-ingénierie de programmes compilés sont nombreux et bien souvent propres à chaque contexte d'analyse. Cependant, nos travaux ont été guidés par cinq défis que l'on retrouve régulièrement lors de l'analyse de programmes compilés :

D1. Prise en main des outils : Le temps accordé aux missions clients est une contrainte forte, ce qui limite grandement l'apprentissage et l'expérimentation de nouveaux outils.

Problématique développée en Section 2.3.11.

D2. Taille des programmes : L'analyse de gros programmes pose régulièrement des soucis aux outils d'analyse automatique. Ces derniers passent rarement à l'échelle (en termes de consommation mémoire et de temps d'analyse). L'analyse manuelle de gros programmes peut également représenter un défi méthodologique pour l'analyste. Ce défi est d'autant plus important quand les missions sont de courte durée.

Problématique développée en Section 2.3.9.

D3. Côtoyer l'assembleur : L'aisance à lire et comprendre de l'assembleur représente des défis méthodologique, syntaxique et de conceptualisation. L'intensité de ces défis est propre à chacun et ne peut s'améliorer qu'avec de l'expérience. Cette expérience doit être d'autant plus importante quand on côtoie différentes formes d'assembleur (ex. SPARC, MIPS, ARM64, i386, etc.).

Problématique développée en Section 2.3.3.

D4. Compréhension du flot de contrôle : Comprendre le flot de contrôle d'un programme est une tâche importante dans un processus d'analyse car cela permet d'avoir une vision claire de ce que fait (sémantiquement) le programme. Par exemple, un analyste peut être amené à vouloir connaître quelles sont les valeurs que peut contenir un registre sachant que ces valeurs sont contraintes par le flot de contrôle. Il peut également être amené à vouloir identifier tous les chemins qui mènent à une instruction spécifique. Pour cela, la compréhension du flot de contrôle est donc nécessaire. Cette compréhension est d'autant plus compliquée quand le programme ciblé traite des données complexes, ou bien est obscurci, ou bien est de taille conséquente.

Problématique développée en Section 2.3.5.

D5. Compréhension du flot de données : Tout comme la compréhension du flot de contrôle, la compréhension du flot de données est toute aussi importante. Il existe des situations où l'analyste pourrait avoir besoin d'identifier l'origine d'un calcul dont le résultat est contenu dans un registre ou une cellule mémoire. Pour cela, la compréhension du flot de données est nécessaire et se complique quand le programme ciblé traite des données complexes, ou bien est obscurci, ou bien est de taille conséquente.

Problématique développée en Section 2.3.6.

Dans le Chapitre 2 nous décrivons plus en détail les différentes cibles, contraintes et problématiques que peuvent rencontrer les industriels lors de l'analyse de programmes par rétro-ingénierie. Guidé par les défis $D1$ à $D5$, ainsi que mes différentes expériences antérieures [2], c'est tout naturellement que l'idée de développer un cadre de travail (*framework*) s'est imposé. Les lignes directrices qui ont guidé la conception et l'implémentation du cadre de travail afin de répondre aux précédents défis sont les suivantes :

- ⋆ ***Framework* adapté aux contraintes industrielles (D1+D2)** : La conception du cadre de travail doit pouvoir permettre de s'interfacer rapidement et facilement avec n'importe quel autre outil afin de minimiser le coût d'adaptation de l'analyste à un nouvel outil (voir Section 3.3). Travaillant principalement sur des programmes de taille conséquente et sachant qu'il est difficile de pouvoir analyser un gros programme dans sa globalité, le cadre de travail a été développé de façon à pouvoir analyser (si l'analyste le souhaite) uniquement certaines parties de la cible avec ou sans contexte initial. Bien évidemment, une absence de contexte implique une imprécision des analyses mais ne doit pas être une limitation à l'utilisation du cadre de travail (voir Section 3.3.1). Enfin, le cadre de travail doit pouvoir analyser des programmes de plusieurs millions d'instructions, pour cela des optimisations ont été développées (voir Section 3.6.1 et 3.6.2).

- ⋆ **Représentation intermédiaire (D3)** : Compte tenu de la problématique concernant l'aisance à lire et comprendre de l'assembleur et des contraintes de temps lors des missions (voir Section 2.2, noter que chaque analyste prend plus ou moins de temps à atteindre ses objectifs en vu de son expérience avec l'assembleur), nous avons choisi de représenter la sémantique opérationnelle d'un jeu d'instructions dans une représentation intermédiaire. Cela permet (1) de faciliter la lisibilité et la compréhension du code machine, (2) d'avoir une représentation canonique commune à toutes formes d'assembleur,
(3) d'avoir la possibilité de parcourir et de manipuler les expressions sémantiques sous une forme syntaxique structurée, (4) d'avoir la possibilité d'interpréter les expressions syntaxiques sur un raisonnement purement symbolique (voir $D4$).

- ⋆ **Moteur d'exécution symbolique (D4)** : Afin d'aider à la compréhension du flot de contrôle, nous avons développé un moteur d'exécution symbolique dynamique (voir Section 3.5.3) permettant de représenter le flot de contrôle d'une exécution sous une forme symbolique. Par exemple, cette

2. Pratiquant la rétro-ingénierie depuis plus de 10 ans, travaillant à Quarkslab avant le début de la thèse et effectuant cette dernière au sein de Quarkslab.

représentation peut être utile pour la recherche de vulnérabilités en effectuant de la couverture de code et d'état de façon automatique (mutations symboliques des entrées du programme).

- ⋆ **Moteur d'analyse de teinte (D5)** : Afin d'aider à la compréhension du flot de données, nous avons développé un moteur de teinte dynamique (voir Section 3.5.2) permettant le suivi des données au cours d'une exécution. Le moteur de teinte guide également le moteur symbolique pour simplifier certaines expressions symboliques afin de répondre au défi $D2$ (voir Section 3.6.2).

Le cadre de travail est présenté sous la forme d'une bibliothèque de développement (voir Chapitre 3) afin d'aider les analystes au mieux durant leurs missions et non comme un outil clef en main qui limiterait le champs des possibilités. Nous avons retenu ce choix de conception car nous savons par expérience qu'il est très difficile de créer un outil qui répond à tous les besoins des différentes missions compte tenu des cibles d'analyses (voir Section 2.1) et des contraintes de missions (voir Section 2.2).

Le cadre de travail est présenté ainsi que formalisé dans le Chapitre 3 et des exemples de scénarios sont présentés dans le Chapitre 4. Enfin, une étude montrant l'efficacité et la généricité du cadre de travail contre des programmes obscurcis est présentée dans le Chapitre 5.

1.4 Les contributions de la thèse

Nos travaux sont organisés autour des 3 contributions principales suivantes :

1. **La rétro-ingénierie et ses problématiques** : Nous collectons et partageons les problématiques réelles des industriels lors de l'analyse de programmes binaires. Pour cela nous avons effectué une enquête auprès d'experts issus de différents milieux industriels et nous discutons des différents apports potentiels en termes d'automatisation et d'outillage pour aider les analystes à résoudre certaines de ces problématiques. Les différentes cibles, contraintes et problématiques qui sont ressortis de cette enquête sont présentées dans le Chapitre 2.

2. **Formalisation d'un cadre de travail** : Le développement du cadre de travail (guidé par les problématiques des industriels) a été initié un an avant le début de la thèse et est utilisé au sein de Quarkslab dans des missions industrielles réelles. Cette thèse apporte une contribution de formalisation des composants internes ainsi que des analyses mises en place dans le cadre de travail. La formalisation de ces composants permet de faciliter la comparaison entre nos analyses et des analyses issues d'autres projets.

3. **Dévirtualisation binaire** : Nous avons publié [82, 83] une approche complètement automatique permettant la dévirtualisation de code binaire en combinant les fonctionnalités de Triton. De plus, nous discutons des limitations et des garanties de notre approche puis nous démontrons le potentiel de la méthode en résolvant automatiquement le *challenge* Tigress (du moins la partie *non-jitted*) d'une manière complètement automatisée [3]. Nous mettons également en place un banc d'expérimentation permettant d'évaluer notre approche sur plusieurs formes de combinaisons de protection. Enfin, nous fournissons un cadre de travail *open-source* [4] afin de (1) reproduire nos résultats (2) ajouter des nouveaux exemples de programme à protéger ainsi que de nouvelles formes de protection pour évaluer différentes attaques.

3. `http://tigress.cs.arizona.edu/solution-0004.html`
4. `https://github.com/JonathanSalwan/Tigress_protection`

1.5 Le plan du manuscrit de la thèse

Le **Chapitre 2** est une étude menée auprès des industriels afin de recenser les différentes cibles, contraintes et problématiques que peuvent rencontrer les analystes lors d'une rétro-ingénierie de programme compilé. Cette étude a pour but d'exposer ce qui est récurrent pour les analystes afin de mieux comprendre leurs besoins.

Le **Chapitre 3** présente le cadre de travail Triton afin d'apporter une aide aux analystes dans le processus de rétro-ingénierie. Plusieurs optimisations y sont décrites afin de converger au mieux vers un passage à l'échelle lors de l'analyse de programmes industriels.

Le **Chapitre 4** est un partage d'expérience lors de vraies missions d'audit de sécurité et introduit l'usage de Triton en situation réelle. Ce chapitre a pour but de donner des exemples d'utilisation de Triton et de montrer comment ce dernier est réellement utilisé au sein de Quarkslab.

Le **Chapitre 5** est une étude approfondie sur l'analyse de programmes obscurcis. Il met en avant une nouvelle approche permettant le dé-obscurcissement automatique de code binaire ainsi que l'usage de Triton dans le processus d'analyse. Cette étude a été faite sur des cas d'étude en scénario contrôlé ainsi qu'en situation réelle par le biais d'un *challenge* public dont certains niveaux n'avaient pas été résolus.

Le **Chapitre 6** conclut la thèse autour d'une discussion sur l'ensemble de nos résultats ainsi que sur les ouvertures que la thèse apporte.

Chapitre 2

Cibles, contraintes et problématiques en rétro-ingénierie

Afin de mieux comprendre les contraintes, les problématiques et les défis de l'industrie face à la rétro-ingénierie, nous avons demandé à plusieurs analystes, issus de différentes entreprises, quelles étaient leurs cibles d'analyse puis quelles étaient leurs contraintes de mission d'audit et enfin quelles étaient les problématiques et défis récurrents rencontrés pendant leurs missions. Ce sondage est également complété par mon expérience personnelle et professionnelle (de plus de 10 ans) sur les sujets de la rétro-ingénierie, la recherche et l'exploitation de vulnérabilités.

2.1 Les cibles d'analyse

Les principales catégories de missions pour une société de sécurité informatique effectuant des audits d'applications sont :

— **La recherche de vulnérabilités (erreurs d'implémentation)** : *Un programme possède-t-il des vulnérabilités ? Lesquelles ? Sont-elles difficiles à trouver ? Sont-elles difficiles à exploiter ? Quelle est leur criticité ? Comment les corriger ?*

— **La vérification des bonnes pratiques (erreurs logiques)** : *Un programme utilise-t-il les bonnes API (ex. API cryptographiques pour une appli-*

cation de chiffrement) ? Cette plateforme utilise-t-elle la bonne méthodologie pour identifier ses clients (ex. choix de conception) ?

— **La recherche d'informations** : *Est-ce que le binaire fait bien ce qu'il est censé faire ? Permet-il des fuites d'informations ? Un programme protège-t-il sa propriété intellectuelle ? Certaines informations embarquées dans le binaire sont-elles facilement accessibles (ex. clefs de chiffrement / déchiffrement pour la gestion des droits numériques (DRM)) ? Quel est le coût en termes de temps et de moyen pour trouver ces informations ?*

Ces catégories représentent une grande partie des missions au quotidien mais cela n'exclut pas l'existence d'autres missions plus spécifiques en fonction des contextes des clients.

La cible et son environnement d'analyse peuvent être divers tels que des applications sur *smartphones* ou *desktop*, des *firmwares* de matériels réseau tels que des routeurs, des serveurs ou encore des *Set-Top Boxes* (STB), mais aussi des applications embarquées sur *boards* propriétaires. Les architectures analysées sont également variées telles que les processeurs Intel, ARM, MIPS ou totalement propriétaires (souvent le cas pour les *Electronic Control Unit* (ECU) dans le domaine automobile).

2.2 Les contraintes des missions

Les contraintes de travail sont propres à chaque mission et sont définies par le client mais dans la majorité des cas nous pouvons retenir les contraintes suivantes :

— **Contraintes de temps** : Les temps des missions sont variables et peuvent aller de trois jours à un an et plus. Le temps de la mission est défini par (1) l'objectif de la mission : un nombre de jours est estimé par la société en charge de l'audit et est ensuite discuté avec le client, (2) les moyens financiers du client : dans une grande partie des cas, c'est principalement ce dernier point qui fixe la durée de la mission et l'adaptation de l'objectif initial.

— **Contraintes de support et de matériel** : Le support de travail est défini par le client et l'objectif de la mission. Certains clients donnent uniquement le binaire pour voir ce que peut faire un attaquant en situation réelle ou bien pour des raisons de propriété intellectuelle (audit dit en *black-box*). En revanche certains clients donnent les sources, facilitant la tâche de l'analyste dans la compréhension du code (audit dit en *white-box*). Il arrive également

que des clients donnent le binaire à auditer accompagné des en-têtes du code source ou d'une documentation technique (ex. spécification d'un protocole). Les en-têtes ou documentations techniques fournissent des informations sur les prototypes de fonctions et les structures facilitant la compréhension de certaines parties du code sans pour autant divulguer toute la propriété intellectuelle.

Dans le cas d'un audit sur un binaire sans les sources, l'architecture du binaire (ex. `Intel x86`, `AArch64`, ou encore propriétaire, etc.) rentre dans les contraintes d'une mission. En effet, pour pouvoir analyser un binaire dans de bonne conditions il faut une suite d'outils supportant cette architecture (ex. désassembleur, débogueur) ; dépourvus de tels outils, l'analyste peut se retrouver bloqué face au challenge d'analyser du code binaire *à main nue*. Le temps imparti pour une mission rend bien souvent impossible le développement d'outils ce qui contraint l'analyste à utiliser ce qu'il a à sa disposition et dont l'utilisation reste compatible avec la durée de la mission. Par exemple, sur une mission de courte durée, un analyste ne peut pas se permettre d'apprendre à utiliser une application tierce sans garantie de résultat. L'analyste se limite donc dans de tels cas à l'utilisation d'outils qu'il connaît bien tels que IDA [54] ou GDB [89].

2.3 Les problématiques et les défis récurrents

Pour mieux comprendre les difficultés de la rétro-ingénierie dans le monde de l'industrie, nous avons demandé aux analystes quels étaient les problématiques et les défis récurrents lors de leurs missions. Ceux qui reviennent le plus souvent sont discutés dans les sections suivantes (sans ordre particulier). Le but étant (1) de partager les difficultés concrètes du monde de l'industrie lors d'une rétro-ingénierie de programmes compilés, (2) de trouver des solutions en termes d'outils et de méthodes d'analyse pour aider les industriels dans leurs missions.

Le sondage a été effectué auprès de 30 personnes issues de différents milieux (étudiant, académique, privé) et de différentes structures (école, individuel, petite PME, grand groupe multinational). Lors de sondage, 48.3% des personnes interrogées ont une expérience de moins de 5 ans, 10.3% entre 6 et 10 ans, 27.6% entre 11 et 20 ans et 13.8% qui ont plus de 20 ans d'expérience en rétro-ingénierie.

2.3.1 Le manque d'informations

2.3.1.1 Les informations de *debug*

La recherche d'informations de *debug* est l'une des premières pistes envisagées par un analyste. En effet la rétro-ingénierie est une méthode d'analyse manuelle permettant de comprendre les fonctionnalités d'un code binaire. Toute information permettant de mieux comprendre le comportement de ce code est donc utile. Généralement un analyste recherche en premier lieu les chaînes de caractères car ces dernières peuvent être très explicites. Par exemple le Listing 2.1 montre un extrait de code faisant référence à une chaîne de caractères pour un message d'erreur. Cette chaîne de caractères nous informe dans un premier temps que le nom de la fonction est *newSymbolicExpression* et que son premier argument doit être de type *AstNode* ce que aide grandement à la compréhension du code.

Listing 2.1 – Référence sur une chaîne de caractères

```
.text:20F5D0
.text:20F5D0 loc_20F5D0:
.text:20F5D0 mov     rax, cs:PyExc_TypeError_ptr
.text:20F5D7 lea     rsi, aNewsymbolicexp ; "newSymbolicExpression(): Expects
                                              a AstNode as first argument."
.text:20F5DE mov     rdi, [rax]
.text:20F5E1 xor     eax, eax
.text:20F5E3 call    _PyErr_Format
```

L'absence de chaîne de caractères et des symboles dans un programme rend la rétro-ingénierie moins confortable et accroît le temps d'analyse. En effet, les chaînes de caractères peuvent donner des informations telles que les noms des fonctions, les noms de certains attributs, le comportement attendu, une confirmation explicite de ce qui vient d'être fait, etc.

2.3.1.2 Les *cross-references*

Les *cross-references* sont des références dans le code binaire à des fonctions, des chaînes de caractères ou des symboles connus. Dans la plupart des formats de fichier binaire tels que ELF, PE ou Mach-O les données sont séparées du code (dans des sections ou segments différents).

Prenons comme exemple un échantillon de code issu de l'utilitaire `zip`, le Listing 2.2 est la vue de cet échantillon désassemblé avec Objdump[1]. À l'adresse

1. Autre désassembleur de la suite *GNU Binutils*

`0xe3a7` une référence est faite par offset. En utilisant Objdump, l'analyste doit aller voir à cette adresse (`0x22fa7`, dans la section de données) pour connaître ce qui est référencé. Le Listing 2.3 représente le même échantillon de code mais cette fois désassemblé avec IDA. IDA est un outil permettant de désassembler, de déboguer et de scripter des analyses automatiques sur une grande variété de binaires. La vue du code désassemblé par IDA permet de voir ces *cross-references*. Quand un programme fait référence à un symbole par offset, IDA essaie de résoudre ce symbole et l'affiche au mieux pour l'analyste. Comme on peut le voir, IDA résout la *cross-reference* et affiche directement la chaîne de caractères (ce qui est un gain de temps).

Listing 2.2 – Échantillon de code avec *Objdump*

```
e3a7:   lea     rcx,[rip+0x14bf9] # 22fa7 <_fini@@Base+0x1f3>
e3ae:   mov     edx,0x3e9
e3b3:   mov     esi,0x1
e3b8:   mov     rdi,rbx
e3bb:   call    68a0 <__sprintf_chk@plt>
```

Listing 2.3 – Échantillon de code avec *IDA*

```
.text:000000000000E3A7   lea     rcx, aFoundUnzipVers ; "Found UnZip version %4.2f"
.text:000000000000E3AE   mov     edx, 3E9h
.text:000000000000E3B3   mov     esi, 1
.text:000000000000E3B8   mov     rdi, rbx
.text:000000000000E3BB   call    ___sprintf_chk
```

Le problème sous-jacent à la résolution des *cross-references* est que les adresses de chargement des segments doivent être connues. Reprenons l'exemple du Listing 2.2. L'instruction `lea rcx,[rip+0x14bf9]` calcule l'adresse de la référence en utilisant l'offset `0x14bf9` par rapport à l'adresse de la prochaine instruction (`0xe3ae`). Le calcul `0x14bf9` + `0xe3ae` donne bien `0x22fa7` et cette adresse pointe bien dans une zone de données. Dans cette exemple la résolution peut être faite car l'adresse de chargement de la zone de données est connue du désassembleur (information issue du format de fichier binaire (ex. ELF) qui est nécessaire pour charger le programme). En revanche, lors des missions sur du matériels embarqués, l'analyste possède rarement le programme dans son intégralité. Ce dernier est souvent récupéré (par portion de code) directement en mémoire par le biais de vulnérabilités exploitées (ex. fuite mémoire) ou mécanismes de *debug* (ex. JTAG). En l'absence des en-têtes de format de fichier binaire, les adresse de chargement des zones de code et de données sont donc inconnues et la résolution des *cross-references* ne peut pas se faire de façon automatique par les outils tels que IDA.

La solution appliquée par les analystes pour résoudre ce problème est de deviner les adresses de chargement des ces zones. Pour cela il faut arriver à aligner toutes

les références sur des données cohérentes (cohérentes par rapport au comportement du programme déterminé par l'analyste).

2.3.2 Distinguer les zones de DATA et de CODE

Pour pouvoir analyser le comportement d'un programme par rétro-ingénierie, il faut dans un premier temps identifier quelles sont les zones qui contiennent les instructions du programme (zone dite de *CODE*) et les zones qui contiennent les données (zone dite de *DATA*). Uniquement en regardant ces deux zones, il est très difficile de les identifier (voir le Listing 2.4 et 2.5). C'est en déterminant le rôle de ces zones qu'il est possible de les interpréter. Par exemple, pour une zone de *CODE* nous allons pouvoir désassembler les octets présents dans celle-ci pour avoir une vue syntaxique (désassemblée) des instructions (en partant du principe que l'on connaît l'architecture, ce qui n'est pas toujours le cas).

Listing 2.4 – Extrait d'une zone de *CODE*

```
fe ff ff 48 63 15 b6 75  20 00 29 d3 48 63 db 48
83 fb 01 0f 87 54 07 00  00 41 0f b6 cd 41 0f b6
c4 01 c8 41 0f b6 ce 01  c8 83 e8 01 0f 8f 9f 07
00 00 44 89 f1 44 09 e9  45 84 e4 75 08 84 c9 0f
84 94 01 00 00 48 85 db  0f 84 ac 00 00 00 48 8b
5c d5 00 80 3b 00 0f 84  99 07 00 00 48 8d 35 bd
```

Listing 2.5 – Extrait d'une zone de *DATA*

```
08 00 00 00 00 00 00 00  44 00 00 00 4c 0c 00 00
40 de ff ff 65 00 00 00  00 42 0e 10 8f 02 42 0e
18 8e 03 45 0e 20 8d 04  42 0e 28 8c 05 48 0e 30
86 06 48 0e 38 83 07 4d  0e 40 72 0e 38 41 0e 30
41 0e 28 42 0e 20 42 0e  18 42 0e 10 42 0e 08 00
14 00 00 00 94 0c 00 00  68 de ff ff 02 00 00 00
```

Les principaux fichiers exécutables des différents systèmes d'exploitations tels que PE pour Windows, ELF pour Linux ou Mach-O pour OS X sont des formats de fichier binaire. Ces fichiers contiennent le code à exécuter et les données utilisées et sont également constitués d'en-têtes permettant de définir clairement quelles sont les zones prévues pour les données et quelles sont celles prévues pour le code ainsi que leurs tailles, leurs types, leurs droits, etc. C'est en se basant sur ces informations que les outils tels que IDA ou objdump peuvent fonctionner de manière automatique.

Les problèmes rencontrés : Identifier les zones qui contiennent le code et celles qui contiennent les données est facile quand nous sommes en possession d'un format

de fichier et de ses spécifications (ELF, PE et Mach-O sont bien documentés). Cependant lors de missions sur du matériel embarqué il arrive de (1) tomber sur des formats de fichier propriétaires et non documentés (2) utiliser des fuites mémoires (*Memory Leak Vulnerability* [2]) pour récupérer du contenu aléatoire (3) utiliser des mécanismes de *debug* pour récupérer de la mémoire tels que l'utilisation du *JTAG*, du *SWD* sur ARM, du *st-link v2* sur STM32, lire la puce de Flash qui contient le *firmware* ou encore intercepter les octets entre la puce de Flash et le CPU.

Que ce soit en exploitant une vulnérabilité ou en utilisant une fonctionnalité de *debug*, nous ne savons généralement pas ce que nous récupérons : *Est-ce des données ? Est-ce du code ? Si c'est du code, quelle est l'architecture ? Est-ce un firmware complet ou partiel ? Ou est-ce tout simplement rien d'important ?*

Ces questions font partie intégrante de la démarche d'analyse de matériel embarqué et y répondre est une tâche longue et difficile mais obligatoire. Sans ça, il n'est pas possible de commencer une rétro-ingénierie du contenu présent sur le matériel embarqué.

2.3.3 Côtoyer l'assembleur

Les défis concernant la lecture et la compréhension de l'assembleur sont des sujets très peu discutés. Nombreuses sont les personnes qui ne savent même pas que l'assembleur peut être lu : on entend certains clients dire qu'une fois leur code source compilé il n'est plus compréhensible et donc *protégé*, affirmation qui est évidemment fausse. Il existe des analystes qui lisent et comprennent l'assembleur de façon très claire, et par conséquent, arrivent à comprendre ce que fait sémantiquement un programme. Cependant cette expertise s'acquiert au fil de la pratique et est soumise à certains défis qui se combinent avec ceux présentés dans les autres sections de ce chapitre :

— **Défi méthodologique** : Lors de l'analyse de gros programmes sur un temps de mission très court, il est capital d'avoir une bonne méthodologie d'analyse. Sans une bonne méthodologie, les objectifs de missions seront difficiles à atteindre dans le temps imparti. Que ce soit pour de l'assembleur ou pour tout autre langage, la revue de code nécessite d'être méthodique. *Par où commencer ? Qu'est-ce que je cherche ? Comment y parvenir sans perdre trop de temps ? De quoi vais-je avoir besoin ? . . .*

— **Défi syntaxique** : Afin de comprendre l'assembleur il faut, comme tout

2. `https://www.owasp.org/index.php/Memory_leak`

autre langage, en connaître sa syntaxe et sa sémantique. Cependant, à l'inverse d'un code source, il existe différentes syntaxes et sémantiques qui sont propres à chaque processeur ce qui complique son apprentissage.

— **Défi de conceptualisation** : L'assembleur est un langage dit *de bas niveau* car sa représentation syntaxique est au plus proche de ce qu'exécute réellement le processeur et son paradigme de programmation est très restreint. Lorsqu'un développeur développe un programme, il peut être amené à décrire le fonctionnement de celui-ci avec des concepts dit *de haut niveau* (tel que des classes objets décrites en C++). Une fois compilé, la représentation haut niveau (ex. C++) de ces concepts est transformée en une représentation plus bas niveau (assembleur) et par conséquent la logique haut niveau du développeur est plus compliquée à retrouver lors d'une rétro-ingénierie.

2.3.4 Identification d'algorithmes connus

Un analyste en rétro-ingénierie passe la plus grande partie de son temps à essayer de comprendre ce que fait telle ou telle fonction pour avoir une vision globale du comportement du programme. Suivant les objectifs de l'analyse toutes les fonctions n'ont pas besoin d'être analysées car celles-ci peuvent être (1) inutiles pour l'objectif de la mission, (2) déjà connues et documentées. La question qui revient souvent lors de l'analyse de code sans les symboles est donc : *Comment savoir si la fonction qu'on analyse n'est pas déjà connue et documentée ?* Afin de mieux comprendre la difficulté qui se cache derrière cette question, nous allons parler des différents modes de lien entre les bibliothèques lors de la compilation.

2.3.4.1 Lien statique et lien dynamique des bibliothèques

Une grande partie des programmes développés en `C` et `C++` utilisent des bibliothèques externes (telles que la *libc*, *boost*, etc.). Ces bibliothèques peuvent être liées au programme de façon statique à la compilation ou dynamique lors d'exécution.

Les bibliothèques liées *dynamiquement* à un programme sont indépendantes de ce dernier et sont situées dans le système de fichiers. Quand le programme s'exécute, ce dernier s'occupe de charger les bibliothèques utilisées dans une zone mémoire prévue à cet effet puis un composant tiers du système (c'est le rôle de

`ld`[3] sous Linux) s'occupe d'initialiser la table des appels externes pour lier chaque appel à son code situé dans les bibliothèques.

Les bibliothèques liées *statiquement* à un programme sont embarquées dans ce dernier à la compilation pour ne former qu'un seul binaire contenant toutes les dépendances nécessaires au fonctionnement de celui-ci. Il n'existe donc pas d'appel externe (à l'exception des appels système).

	Lien Statique	**Lien Dynamique**
Avantage	Aucune dépendance sur le système	Taille du binaire réduite
Inconvénient	Taille du binaire augmentée	Dépendances sur le système

TABLE 2.1 – Avantages et inconvénients des bibliothèques liées statiquement ou dynamiquement

Les avantages et les inconvénients des bibliothèques liées statiquement ou dynamiquement sont illustrés dans le Tableau 2.1. Sur certains systèmes (tels que les routeurs) nous allons plutôt retrouver des binaires qui sont liés dynamiquement à leurs bibliothèques car (1) les constructeurs maitrisent tout l'écosystème ainsi que la chaîne d'exécution et ne risquent pas de rencontrer des problèmes de dépendances (2) cela permet de réduire au minimum l'espace utilisé pour le bon fonctionnement du système. À l'inverse on peut voir des programmes avec des bibliothèques liées statiquement dans le milieu de l'édition de logiciels propriétaires. En effet, les éditeurs ne connaissant pas les bibliothèques disponibles sur les machines de leurs clients, ils préfèrent donc embarquer dans leur logiciel toutes les dépendances nécessaires pour son bon fonctionnement. Cela évite les problèmes liés au manque de dépendance. Cela est également vrai dans le milieu de l'édition de *malware* où le but est de s'exécuter quoi qu'il arrive.

2.3.4.2 Les problèmes rencontrés

Identification d'algorithmes connus : Quand un binaire est lié statiquement à ses bibliothèques, le code de ces dernières est donc embarqué dans le programme. La plupart des éditeurs suppriment les symboles et toute autre information permettant de mieux comprendre le programme afin de conserver leur propriété intellectuelle. Cela signifie que nous n'avons plus aucune information permettant de distinguer le code qui a été écrit par l'éditeur logiciel et le code qui est propre aux bibliothèques (qui elles sont connues et documentées).

3. `https://manpages.debian.org/stretch/manpages/ld-linux.8.en.html`

Listing 2.6 – Échantillon de code

```
int main(int ac, const char *av[]) {
  if (ac != 2)
    return 0;
  return atoi(av[1]);
}
```

Prenons comme exemple le code illustré par la Listing 2.6. Si nous compilons ce code et que nous lions ses bibliothèques dynamiquement nous voyons clairement que ce programme fait appel à une fonction externe nommée `atoi` (voir le Listing 2.7). Notons que le code la fonction `atoi` n'est pas dans le programme mais est chargé dynamiquement lors de l'exécution du programme.

Listing 2.7 – Échantillon de code

```
.text:0000000000000636    mov     rax, [rbp+var_10]
.text:000000000000063A    add     rax, 8
.text:000000000000063E    mov     rax, [rax]
.text:0000000000000641    mov     rdi, rax          ; nptr
.text:0000000000000644    call    _atoi
```

Si nous compilons ce même programme et lions ses bibliothèques statiquement, le code de la fonction `atoi` sera embarqué dans le binaire. La Figure 2.1 montre l'arbre d'appels (*Call Graph*) fourni par IDA. Nous pouvons voir que la fonction `main` fait appel à la fonction `atoi`, qui elle-même, fait appel à la fonction `strtol`. Le fait d'avoir les symboles permet à l'analyste d'identifier rapidement le rôle d'une fonction (en partant du principe que son nom n'a pas été modifié par une protection logicielle).

Reprenons le même exemple du Listing 2.6 toujours en liant les bibliothèques statiquement mais cette fois-ci en enlevant tous les symboles du programme compilé. La Figure 2.2 illustre exactement le même arbre d'appels que la Figure 2.1 mais cette fois-ci sans les symboles. Dans cette situation l'analyste peut difficilement savoir au premier coup d'œil que la fonction `main` fait appel à la fonction `atoi`, qui elle-même, fait appel à la fonction `strtol`. L'analyste va devoir analyser ces fonctions pour en déduire leurs rôles. Cela implique donc une perte de temps non négligeable dans le temps imparti d'une mission sachant que ces fonctions n'ont pas été écrites par l'éditeur et que leur comportement est connu et documenté. Notons que la plupart des binaires audités ne possèdent pas de symbole de *debug*.

Sur certaines missions il arrive de devoir auditer des programmes liés statiquement à leurs bibliothèques sans symbole et protégés contre la rétro-ingénierie par des protections logicielles. Ces protections modifient la structure du code pour la rendre plus complexe à analyser (ex. VMProtect [4], Themida [2], O-LLVM [58],

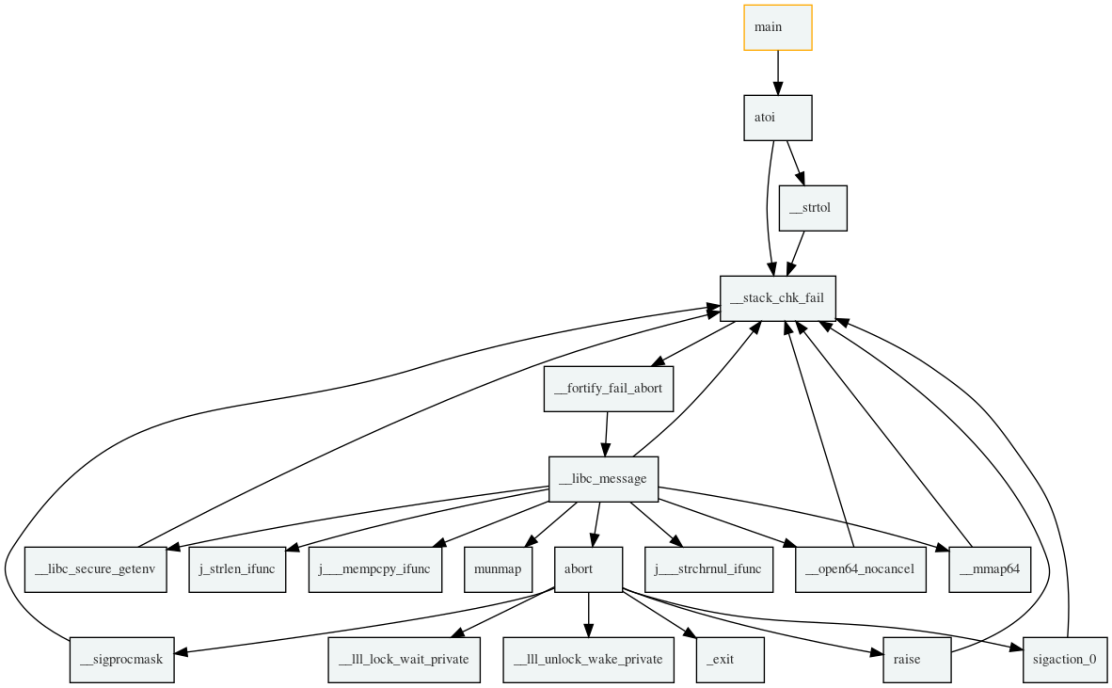

FIGURE 2.1 – Arbre d'appels fourni par IDA du Listing 2.6
avec ses bibliothèques liées statiquement avec les symboles

etc.). L'analyste se retrouve à devoir analyser des fonctions obscurcies, pourtant connues et documentées, ce qui est un gaspillage énorme en termes d'effort et de temps.

Pour reconnaître certaines fonctions, IDA propose un mécanisme de signature de fonctions connues (principalement les bibliothèques standards) appelé *FLIRT* [4]. Les informations requises par *FLIRT* sont conservées dans un fichier de signatures fourni par l'éditeur Hex-Rays et modifiable par les analystes. Chaque fonction est représentée par un motif. Le modèle de reconnaissance (la signature) est les 32 premiers octets d'une fonction où tous les octets variants sont marqués par deux points (“..”) et les octets invariants par leur valeur en hexadécimal. Le Listing 2.8 est un extrait de ce fichier de signatures illustrant les signatures de 4 fonctions.

Listing 2.8 – Exemple de signature FLIRT

```
558BEC0EFF7604..........595995DC3558BEC0EFF7604..........595995DC3  _registerbgidriver
558BEC1E078A66048A460E8B5E108B4E0AD1E9D1E980E1C0024E0C8A6E0A8A76  _biosdisk
558BEC1EB41AC55604CD211F5DC3....................................  _setdta
558BEC1EB42FCD210653B41A8B5606CD21B44E8B4E088B5604CD219C5993B41A  _findfirst
```

4. https://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml

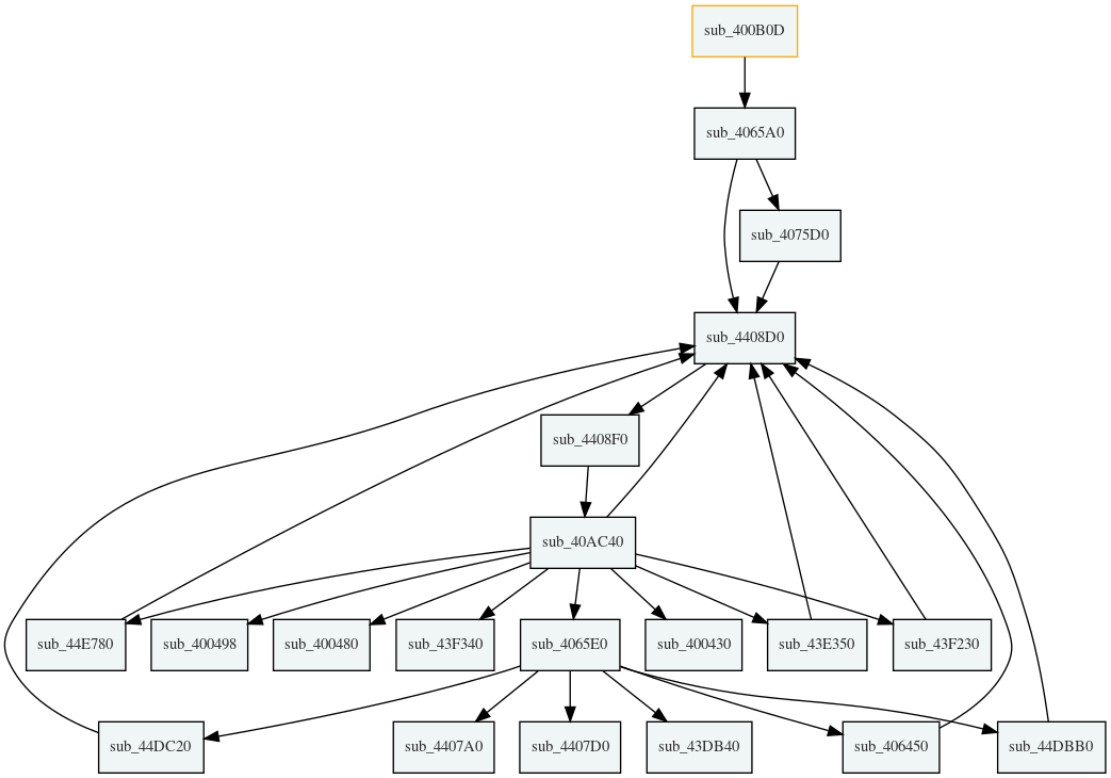

FIGURE 2.2 – Arbre d'appels fourni par IDA du Listing 2.6 avec ses bibliothèques liées statiquement sans les symboles

Ce mécanisme ne fonctionne que si les fonctions ne sont pas obscurcies. Pour résoudre le problème de fonctions connues obscurcies, Camille Mougey a publié un outil baptisé *Sibyl* [6] basé sur le *framework* Miasm [37]. Sibyl voit les fonctions comme des boîtes noires. En injectant des entrées connues aux fonctions, et en connaissant les sorties attendues, on peut en déduire le rôle des fonctions. Par exemple avec le Listing 2.9, en connaissant les entrées et les sorties on peut déduire que la fonction `sub_4075D0` est la fonction `atoi`, et cela peut importe si elle est obscurcie ou non.

Listing 2.9 – Entrées → sorties d'une fonction inconnue

```
sub_4075D0("1") = 1
sub_4075D0("1234") = 1234
sub_4075D0("2738642374") = 2738642374
sub_4075D0("-1") = 0xffffffffffffffff
sub_4075D0("test") = 0
```

Sibyl fournit une série de tests permettant d'identifier des fonctions connues telles que `strlen, strcpy, strcat, md5, sha256, etc.` à partir de leurs entrées

et sorties. Cependant, comme il n'est pas possible de couvrir tout l'espace d'entrée de chaque fonction, Sibyl est donc correct (si la fonction inconnue est dans la base de tests de Sibyl) mais pas complet, ce qui veut dire qu'il n'est pas possible de garantir que la fonction devinée est la bonne. Pour être complet il faudrait décrire tous les états possibles de la fonction, ce qui, dans la pratique n'est pas possible et cela reste donc un problème ouvert pour les analystes.

Reconstruction de paramètres d'optimisation à partir du binaire : Nous pouvons lister un autre exemple mais cette fois-ci avec un code optimisé. Il existe une étude[5] où il fallait analyser une optimisation pour la multiplication dans un corps fini sur des éléments 128 bits pour GCM utilisé dans AES-GCM. Dans cette situation les analystes avaient le résultat compilé d'une optimisation manuelle de la multiplication d'un corps fini, mais aucune publication de sa description n'avait été faite, empêchant toute application ultérieure de l'optimisation avec des paramètres différents dans un autre contexte. Le code en charge de l'optimisation est illustré par le Listing 2.10.

Listing 2.10 – Optimisation de corps fini pour AES-GCM

```
vmovdqa     T3, [W]
vpclmulqdq  T2, T3, T7, 0x01
vpshufd     T4, T7, 78
vpxor       T4, T4, T2
vpclmulqdq  T2, T3, T4, 0x01
vpshufd     T4, T4, 78
vpxor       T4, T4, T2
vpxor       T1, T1, T4   ; result in T1
```

Une des étapes de la mission était de porter cette optimisation sur des éléments de 64 bits au lieu de 128 bits. Pour cela il a fallu extraire les paramètres pertinents et extraire la sémantique : identifier les constantes ainsi que les polynômes puis comprendre comment ils interviennent. Les analystes ont commencé par étudier toutes les optimisations de la multiplication dans un corps fini sur des éléments 128 bits pour AES-GCM qui étaient publiées afin de comprendre les concepts généraux. Durant leurs recherches ils ont trouvé une publication qui nommait une étape principale de l'optimisation (la réduction de Montgomery) et ont pu commencer à se renseigner sur son implémentation. Trouver le nom de l'optimisation ne leur a pris que quelques minutes, cependant le défi était de se convaincre que c'était bien cette optimisation qui était utilisée. Pour en être convaincu, ils ont dû prouver formellement l'optimisation à la main, ce qui leur a pris plusieurs jours. En effet, la preuve arithmétique manuelle est sujette à des erreurs (car effectuée par des humains sur une représentation des données peu intuitive) et dans cette situation

5. https://blog.quarkslab.com/reversing-a-finite-field-multiplication-optimization.html

il était difficile de savoir si c'était la preuve qui était fausse ou alors que le code n'effectuait pas de réduction de Montgomery. Cet exemple montre encore une fois que la reconnaissance d'algorithmes connus (sémantiquement parlant) peut être un atout pour les analystes et des recherches sont menées dans ce sens [64].

2.3.5 Compréhension du flot de contrôle et résolution des contraintes

Peu importe les cibles d'analyse (voir Section 2.1) il existe des situations où un analyste devra résoudre des conditions de branchement pour atteindre un point spécifique dans un programme (ex. atteindre une fonction de déchiffrement, déclenchement d'une vulnérabilité, vérification de fonctionnalités, etc.). Résoudre manuellement les contraintes de branchement lors d'analyses statiques peut devenir une tâche difficile, surtout quand des enchaînements conséquents d'opérations arithmétiques interfèrent avec les conditions de branchement.

Pour la recherche de vulnérabilités, la résolution automatique des contraintes de branchement est un sujet intéressant. Généralement les outils d'analyse automatique pour la recherche de vulnérabilités (type *fuzzer*) essaient de faire muter les entrées d'un programme (ex. lecture de fichiers, environnement système, paquets réseau, etc.) afin de couvrir un maximum d'instructions et ainsi tenter de faire *crasher* le programme. Pour pouvoir résoudre des contraintes de branchement de façon automatique, des analyses ont été mises en place telles que l'exécution symbolique [61, 29, 86, 51, 49, 96, 76].

L'exécution symbolique consiste à représenter un programme par des formules arithmétiques et logiques. Les entrées du programme sont représentées par des variables dites *symboliques*. L'exécution du programme est modélisée par ce qu'on appelle un prédicat de chemin. Ce prédicat de chemin est une formule logique représentant l'ensemble des contraintes de branchement du programme. L'exécution symbolique fournit ensuite des modèles (des valeurs) aux variables symboliques permettant de satisfaire les contraintes de branchement du prédicat de chemin.

Cette méthode prometteuse est à l'origine de plusieurs outils [18, 87, 26, 28, 50, 92] et permet d'effectuer une couverture de code automatique afin de trouver des entrées pouvant amener à un *crash* du programme ciblé et ainsi découvrir de potentielles vulnérabilités.

Les problèmes rencontrés : La résolution manuelle des contraintes de branchement est une méthode fastidieuse et difficilement applicable en cas d'analyse de programmes obscurcis. Concernant l'usage d'outils automatiques, nous pouvons citer plusieurs problèmes :

— la taille des binaires à auditer peut poser des problèmes aux outils d'analyse (voir Section 2.3.9).

— les outils sont difficilement adaptables à toutes les situations et scénarios rencontrés durant les missions et une perte de temps n'est pas envisageable (voir Section 2.3.11).

— bien que l'exécution symbolique soit une méthode prometteuse, elle est actuellement confrontée à deux problèmes importants :

  ⋆ *Passage à l'échelle* : D'une part, le nombre de chemins à explorer explose avec la taille du programme (voir Section 2.3.9), et dès lors la technique (même si elle reste applicable) n'explore qu'une petite partie de l'espace des comportements, et risque même de se " perdre " dans l'exploration de comportements tous plus ou moins similaires (boucles). D'autre part, le fait de devoir construire tout au long de l'exécution les conditions de branchement rend les expressions symboliques trop complexes à résoudre dans certains cas (boucles, appels de fonction récursive, opérations de hachage, etc.).

  ⋆ *Pouvoir de détection des fautes* : Premièrement, l'exécution symbolique est essentiellement dirigée par la couverture du code à analyser pour la recherche de vulnérabilités, ce qui n'est pas forcément l'objectif principal de la mission (voir Section 2.1). Deuxièmement, une couverture de code ne garantit pas de trouver tous les bogues présents sur un chemin de programme donné, puisque d'une part certaines erreurs ne se déclenchent que sur certaines entrées bien précises, et d'autre part certains bogues ne provoquent pas de *crash* immédiat (ex. *use-after-free*, *race condition*, *out-of-bound*, etc.). Ceci peut entraîner une certaine inefficacité de la technique pour la recherche de vulnérabilités.

2.3.6 Compréhension du flot de données et suivi de données

Un thème qui revient souvent quand on questionne les analystes au sujet des problèmes qu'ils rencontrent lors d'une rétro-ingénierie de programme compilés, c'est la compréhension du flot de données lors d'analyses manuelles. Par exemple, isoler toutes les instructions qui ont un lien avec la lecture d'un fichier ou d'une trame réseau afin d'identifier rapidement certaines parties du code qui seraient

potentiellement intéressantes telles que les zones de code en charge du traitement (*parsing*) de contenu (ces zones sont réputées difficiles à bien concevoir et donc potentiellement vulnérables). Afin d'aider à la compréhension du flot de données, des analyses ont été developpées telles que l'analyse de teinte [86, 30, 59]. Cependant cela peut être difficile de trouver un outil qui s'intègre bien dans le contexte de la mission compte tenu de ses contraintes (voir Section 2.2) et des problèmes sous-jacents tels que la taille des programmes ciblés (voir Section 2.3.9) ainsi que la prise en main des outils (voir Section 2.3.11).

2.3.7 Extraction et réutilisation de code binaire

Dans certaines situations il est intéressant pour un analyste de pouvoir extraire et réutiliser du code binaire issu d'un programme propriétaire [6]. Prenons comme exemple un logiciel permettant de lire un contenu multimédia chiffré (ex. une vidéo). Cette vidéo ne peut être lue que par le logiciel propriétaire car celui-ci contient les clefs pour la déchiffrer (principe entre autres, de la gestion des droits numériques ou *Digital Rights Management (DRM)* en anglais). Dans cette situation si l'analyste veut développer un logiciel tiers permettant de visionner le contenu de la vidéo, il peut soit comprendre le mécanisme de chiffrement, récupérer les clefs de déchiffrement et redévelopper un logiciel tiers permettant de visionner les vidéos, soit extraire la partie de code permettant le déchiffrement de la vidéo et embarquer cette partie de code (*code lifting*) directement dans un logiciel tiers. Cela permet d'éviter la compréhension complète du mécanisme de déchiffrement et ainsi gagner du temps d'analyse.

Le principal défi : Il n'y a aucune difficulté à extraire du code binaire si celui-ci n'a aucune dépendance avec d'autres parties du programme. En revanche, la difficulté se manifeste quand il est nécessaire d'extraire du code binaire dont le fonctionnement dépend de plusieurs états mémoire externes à celui-ci. Prenons comme exemple l'échantillon de code du Listing 2.11 et imaginons que l'analyste veut extraire le code de la fonction `f` du programme. Cette fonction accède à un état mémoire externe à son périmètre local via la variable `global_var`. Cette même variable est définie par une autre fonction (`init`) qui n'a pas de lien direct avec la fonction `f` (par lien direct on entend fonction appelée ou appelante). Sur un gros binaire il peut y avoir des milliers d'interactions qui définissent l'état mémoire externe (non constant) d'une fonction que nous voulons extraire. La difficulté est

6. Dans le cadre strict de la mission et encadré par un contrat de collaboration afin d'évaluer la difficulté réelle d'une telle attaque.

de déterminer toutes les dépendances indirectes d'une fonction pour que celle-ci puisse être exécutée et fonctionner dans un contexte tiers. Notons que, dans cet exemple, l'extraction de la fonction `f` avec ses dépendances revient à extraire tout le code du programme et donc tout simplement à utiliser le programme. Cependant, dans le cadre d'un programme propriétaire on considère que l'objectif est d'extraire l'intelligence de celui-ci sans les mécanismes de protection (ex. signature, intégration, etc.), ceci afin de construire un programme tiers avec la même intelligence mais sans restriction.

Listing 2.11 – Échantillon de code

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

static char global_var[32];

void init(char *user_input) {
  for (unsigned int i = 0; i < sizeof(global_var); i++) {
    global_var[i] = user_input[i] ^ 0x55;
  }
}

void f(void) {
  int fd = open("./data", O_CREAT | O_RDWR | O_APPEND, S_IRUSR | S_IWUSR);
  write(fd, global_var, sizeof(global_var));
  close(fd);
}

int main(int ac, char *av[]) {
  if (ac != 2)
    return 0;

  init(av[1]);
  f();

  return 0;
}
```

2.3.8 Stabilité de l'exploitation logicielle

Que ce soit de l'analyse de code source ou binaire, manuelle ou automatique, la recherche de vulnérabilités sur des programmes informatiques a plusieurs buts :

— **Éducatif** : Apprendre et comprendre les différents types de vulnérabilités et les concepts d'exploitation. Développer et comparer les outils d'analyse automatique. Missions de sensibilisation.

— **Défensif** : Identifier et corriger les vulnérabilités pour protéger un programme ou un système informatique. C'est principalement ce type de mission qu'effectue une société d'audit de sécurité.
— **Offensif** : Identifier et exploiter les vulnérabilités pour attaquer, corrompre ou espionner. Ce point est très peu discuté en raison de sa sensibilité juridique mais reste une thématique intéressante pour ses défis et pour rendre plus robustes les défenses.

Dans cette section nous allons discuter du défi que représente la stabilité d'une exploitation logicielle. L'exploitation d'une vulnérabilité est le fait de corrompre le comportement initial du programme en injectant des données que ce dernier traite de façons erronées. Généralement on cherche à faire en sorte que ce mauvais traitement des données cause un *crash* du programme. C'est au moment du *crash* et en regardant l'état des registres et de la mémoire que nous pouvons déterminer si une exploitation est possible. Si c'est le cas, on parle alors de vulnérabilité logicielle, dans le cas contraire, si aucune exploitation n'est possible on parle lors de bogue informatique. Tous les bogues ne sont donc pas forcément des vulnérabilités mais toutes les vulnérabilités sont issues de bogues ou de problèmes plus profonds d'architecture (faille de conception dans un protocole, etc.). Il n'est pas toujours possible de déterminer avec certitude l'exploitabilité ou non d'un bogue et donc de décider si c'est ou non une vulnérabilité.

Challenges principal et sous-jacent : La réussite et la complexité de l'exploitation d'une vulnérabilité dépendent fortement des mécanismes de sécurité rajoutés dans le programme lors de la compilation (ex. *Position-Independent Code and Executable*, *Control-Flow Integrity*, *Stack Canary*, *Pointer Authentication*, etc.) mais aussi mises en œuvre pendant l'exécution par le système d'exploitation (ex. *Address Space Layout Randomization*, *No-eXecute*, *Supervisor Mode Access Prevention*, etc.). Outre le défi principal qui est la complexité de réussir une exploitation en contournant l'ensemble de ces mécanismes de sécurité, il existe un problème sous-jacent lié à la stabilité d'une exploitation sur différentes machines.

Réussir une exploitation sur une machine ne garantit pas que l'exploitation fonctionnera également sur une autre machine. Les exploits sont généralement propres à l'espace d'adressage d'un processus ainsi qu'aux états de la mémoire et des registres au moment du *crash*. Par exemple, si la réussite d'un exploit repose sur une constante placée dans un registre ou un pointeur récupéré dans la mémoire au moment du *crash*, ces valeurs peuvent ne pas être identiques lors d'un *crash* sur une autre machine, auquel cas l'exploitation échouera. Sur les systèmes d'exploitation tels que Linux où les applications sont compilées différemment en fonction

des distributions (ex. version et optimisations du compilateur activées par défaut, version des bibliothèques partagées, fonctionnalités de l'application choisies à la compilation, etc.), la probabilité d'avoir des états différents au moment du *crash* sur différentes machines est augmentée. Sur les systèmes d'exploitation tels que Windows où les applications sont fournies de façons précompilées, la différence d'état au moment du *crash* est diminuée (ce qui facilite l'exploitation). Si on considère que l'exploitation doit être appliquée sur des systèmes d'exploitation tels que Linux et où les applications ne sont pas précompilées, le défi réside dans l'aptitude à faire un exploit sans se reposer sur des états variables. Des solutions ont été proposées pour générer des exploits de façon automatique ([11, 55]) mais ne sont applicables que si on possède l'application ciblée et son contexte, ce qui n'est pas forcément le cas quand on effectue des attaques à distance.

2.3.9 Taille des programmes

Durant les missions d'audit de programmes (peu importe la cible, voir Section 2.1), la taille des binaires à analyser peut dépasser plusieurs dizaines de mégaoctets. Ces tailles assez conséquentes posent plusieurs problèmes :

— **Problème d'outillage** : L'analyse automatique de gros binaires pose régulièrement des problèmes aux outils, ce qui peut les rendre inutilisables ou inadaptés [69, 13, 19, 39]. Une analyse manuelle est donc bien souvent nécessaire.

— **Analyse manuelle, problème de repère** : Lors d'une analyse manuelle sur un gros binaire, l'analyste peu se sentir vite perdu face à cette quantité de code à auditer. Bien souvent il ne sait pas par où commencer et a sans cesse la sensation de passer à coté de quelque chose d'important (voir les défis sur l'habitude de l'assembleur en Section 2.3.3). On peut citer plusieurs méthodologies telles que se baser sur une trace d'exécution afin de déterminer quelles sont les parties de code exécutées, ou alors commencer par identifier des zones de code intéressantes (en accord avec les objectifs de la mission, voir Section 2.1) puis de remonter dans l'arbre d'appels pour voir comment il est possible d'atteindre ces zones. Il existe une infinité de méthodologies qui sont propres à chaque analyste mais dans ce type de situation c'est l'intuition et l'expérience qui priment.

2.3.10 L'analyse statique et le milieu de l'embarqué

D'une manière générale l'analyse statique manuelle sur de gros binaires pose problème pour un analyste en raison du manque de contexte. La combinaison d'analyses statiques avec des analyses dynamiques est donc une solution classique. Par exemple il est courant de commencer une analyse manuelle sur une vue statique des instructions désassemblées (ex. avec IDA) et de faire appel à des débogueurs (tels que GDB ou WinDbg) pour obtenir un contexte à un point intéressant pour l'analyste (ex. état des registres et de la mémoire). Notons également qu'atteindre un point spécifique dans un programme n'est pas toujours une tâche facile (voir Section 2.3.5).

Les problèmes rencontrés : Le problème général est le manque de contexte sur l'analyse manuelle de gros binaire en statique et plus particulièrement dans le milieu de l'embarqué quand il est difficile de pouvoir exécuter le code à auditer (ex. manque d'outil et d'environnement). Prenons un exemple dans le domaine automobile où la présence de CPU tel que le NEC v850 [7] est répandu. On trouve très peu d'outils publics permettant l'émulation ou l'analyse de ce jeu d'instructions ce qui restreint les analystes à une analyse statique manuelle. Sachant que ces architectures sont très peu analysées (car très spécifiques), les outils supportant ces dernières sont donc peu testés et souvent incomplets. Par exemple, IDA ne supporte pas complètement le désassemblage du jeu d'instructions NEC v850 et il est fréquent de devoir développer soit même des *plugins* pour étendre ou corriger le désassemblage des instructions pour NEC V850E1 [8].

Outre le fait que l'analyse statique manuelle de ces architectures exotiques n'est pas confortable pour un analyste (dans le milieu de l'embarqué il existe autant d'architectures que de problèmes sur un bateau [9], ce qui permet rarement de se familiariser avec une architecture particulière), le manque de contexte pose également plusieurs problèmes :

— **Pointeurs sur fonctions** : L'usage de pointeurs sur fonctions est un problème bien connu pour les analyseurs automatiques statiques de codes. Ces outils reposent sur des méthodes d'analyse de plage de valeurs (voir *Value-Set Analysis* [12]) pour déterminer les différentes possibilités de destination des sauts. Cependant ces valeurs sont vite imprécises si la taille du binaire à analyser est conséquente ou si la structure du code possède des propriétés parti-

7. `https://en.wikipedia.org/wiki/V850`
8. `https://github.com/patois/NECromancer`
9. Proverbe pour désigner un grand nombre d'architectures.

culières (ex. boucles, bibliothèques externes, code asynchrone, etc.). Concernant l'analyse statique manuelle sans contexte dynamique, la résolution des sauts indirects est une tâche très difficile et souvent bloquante pour un analyste.

— **Cross-references** : Sans contexte initial et sans format de fichier permettant de connaître l'adresse de base de chargement du code (voir Section 2.3.2), les outils tels que IDA ne permettent pas de résoudre les *cross-references* de façon automatique, ce qui rend la tâche plus difficile pour l'analyste (voir Section 2.3.1).

— **Exploitation en aveugle** : Quand une vulnérabilité est trouvée via une analyse statique, elle ne fonctionne pas forcement sur le matériel en situation réelle. Afin d'en avoir confirmation, l'analyste commence par essayer de déclencher la vulnérabilité sur le matériel en question. Si le matériel répond par un signe distinct comme un *crash* ou un comportement suspect, il est envisageable de dire que la vulnérabilité est bien réelle. Toutefois, l'exploiter sans moyens de *debug* reste un vrai défi. En effet, l'exploitation de vulnérabilité binaire repose souvent sur la connaissance de l'état des registres et de la mémoire au moment du *crash*. Ayant connaissance de ces informations et en adéquation avec le type de vulnérabilité (ex. débordement de tampon sur la pile), il est possible de détourner le flux d'exécution. Sans contexte au moment du *crash* et sans moyen de *debug*, l'exploitation de vulnérabilité binaire à l'aveugle est un vrai défi.

D'après les retours d'expérience des analystes interrogés pour le sondage, les avis sont unanimes sur le fait qu'une analyse statique va de pair avec une analyse dynamique. Cependant, dans certains cas, et notamment sur des missions de recherche de vulnérabilité sur du matériels embarqués, l'usage d'analyse dynamique n'est pas toujours possible ce qui laisse une question ouverte : *Comment simuler rapidement du code afin d'avoir un environnement de debug réduit au minimum ?*

2.3.11 Prise en main des outils dans le temps imparti

La prise en main des outils dans un temps imparti lors d'une mission d'analyse est un vrai défi. Les missions sont fortement contraintes par le temps (voir Section 2.2) et permettent rarement d'avoir la possibilité d'apprendre à utiliser un outil.

Le problème de fond est l'absence de garantie de résultats par les outils. Bien souvent quand on lit les descriptions des outils et leurs exemples d'utilisations, on

a du mal à savoir réellement si :

— **Questionnement sur le choix** : L'outil est le bon choix compte tenu de la problématique de la mission et des objectifs. *Pourquoi cet outil et pas un autre ?*

— **Questionnement sur les fonctionnalités** : L'outil est à un niveau de développement avancé ou stable. *Est-ce un outil mature ou est-il encore au stade de preuve de concept ? Fonctionnera-t-il sur mon programme ciblé ? Quelles sont ses limitations ? Sera-t-il adaptable compte tenu des objectifs de la mission ?*

— **Questionnement sur les résultats** : L'outil nous fournira les résultats attendus. *Aurons-nous les informations que nous recherchons ? Les objectifs seront-ils remplis ? Le client sera-t-il content ?*

Répondre à ces questions est une tâche difficile sans avoir eu d'expérience avec l'outil. Faire le choix d'utiliser un outil inconnu lors d'une mission est donc une prise de risques vis-à-vis des objectifs attendus par le client dans le temps imparti. Par exemple, sur une mission de 10 jours, il n'est pas envisageable de prendre 3 jours pour apprendre à utiliser un outil et de se rendre compte le 4e jour que l'outil ne correspond pas à la mission et ses objectifs. C'est pourquoi les analystes prennent rarement le risque de sortir de leur zone de confort et se limitent à l'usage des outils qu'ils connaissent (le plus populaire étant IDA).

2.4 Conclusion sur l'outillage et les défis

Dans ce chapitre nous avons listé certains défis auxquels les analystes font face lors d'audits de sécurité de programmes compilés.

Dans un domaine en perpétuelle évolution où les programmes informatiques évoluent et se complexifient chaque jour, les outils sont devenus indispensables pour les analystes et font partie intégrante d'une démarche d'audit de sécurité.

Bien que les outils ne soient pas une solution à tous les problèmes, ils peuvent tout de même être d'une grande utilité. Par exemple l'utilisation d'une représentation intermédiaire du code binaire peut aider à la résolution de certains points du défi " Côtoyer l'assembleur ", tel que la résolution du challenge syntaxique que représente la lecture d'un code assembleur.

On constate une évolution des techniques d'analyse (ex. analyse symbolique, analyse de teinte, interprétation abstraite, etc.) et le défi aujourd'hui est d'intégrer

ces analyses modernes dans les processus d'audits de sécurité où les analystes ont tendance à utiliser uniquement un désassembleur et un débogueur.

Chapitre 3

Triton : Cadre d'analyse binaire dynamique

3.1 Les objectifs

Triton [5] a été développé afin de répondre aux défis **D1** à **D5** que nous décrivons dans la Section 1.3 ainsi que pour les objectifs décrits ci-dessous :

- ⋆ **Passage à l'échelle** : Permettre de combiner exécution symbolique et exécution concrète afin d'alléger certaines expressions symboliques. La combinaison de ces deux exécutions est communément appelée exécution concolique [87] (pour : exécution **conc**rète et symb**olique**). Les concepts de base sont présentés dans la Section 3.2.3 et les optimisations en Section 3.6.
- ⋆ **Généricité des analyses** : Concevoir une représentation intermédiaire permettant d'avoir une base commune à différentes architectures pour réutiliser les algorithmes et les analyses. L'architecture de la bibliothèque Triton est présentée en Section 3.3.1.
- ⋆ **Interfaçage entre outils** : Proposer une API simple d'utilisation sur différents langages (ex. C++, Python) afin de pouvoir interfacer Triton avec d'autres outils de la communauté sécurité tels que IDA, Pin, Qemu, etc. Un exemple concret combinant Triton et IDA est présenté dans le Chapitre 4.

Dans ce chapitre nous commençons par introduire le contexte scientifique en Section 3.2. En Section 3.3, nous décrivons le fonctionnement global de la bi-

bliothèque Triton et nous y introduisons ses fonctionnalités. Ensuite, nous décrivons en Section 3.4 le modèle de Triton et en Section 3.5 les 3 principaux composants de la bibliothèque : l'exécution concrète, l'analyse de teinte et l'exécution symbolique. Enfin, nous décrivons en Section 3.6 certaines optimisations apportées à l'exécution symbolique permettant le passage à l'échelle sur l'analyse de gros programmes.

3.2 Contexte scientifique

3.2.1 L'analyse de teinte

Une analyse de teinte dynamique [86, 30] est le fait de marquer (*teinter*) un ensemble de données à un instant t sur une trace d'exécution, puis le moteur de teinte propage cette marque (*teinte*) aux registres et cellules mémoire en tout point du programme en accord avec la sémantique de chaque instruction exécutée.

Trace d'exécution	**État de teinte**
	{esi}
(1) mov eax, esi	{esi, eax}
(2) inc eax	{esi, eax, of, sf, zf, af, pf}
(3) mov ebx, eax	{esi, eax, of, sf, zf, af, pf, ebx}
(4) mov eax, 0xbffffff7	{esi, of, sf, zf, af, pf, ebx}
(5) mov [eax], ebx	{esi, of, sf, zf, af, pf, ebx, @eax:32}
(6) mov edi, [eax]	{esi, of, sf, zf, af, pf, ebx, @eax:32, edi}

Table 3.1 – Exemple d'une analyse de teinte dynamique avec pour teinte initiale le registre `esi`

Le Tableau 3.1 illustre un exemple d'analyse de teinte dynamique. Nous commençons l'analyse avec pour teinte initiale le registre `esi`. Au fur et à mesure que la trace est exécutée, la teinte est propagée en accord avec la sémantique des instructions exécutées. Par exemple, au point du programme `(1)`, la teinte est propagée dans le registre `eax`. Au point de programme `(2)`, l'instruction `inc` modifie les drapeaux ce qui propage la teinte dans ces derniers. Au point de programme `(4)`, le registre `eax` est réécrit avec une donnée non teintée, ce qui supprime `eax` de notre état de teinte. Au point de programme `(5)` une donnée teintée (`ebx`) est placée en mémoire, ce qui propage la teinte sur ces cellules mémoire (`@eax:32` : adresse de `eax` sur 32 bits). Un fois le point de programme `(6)` exécuté, notre état

de teinte est : {`esi, of, sf, zf, af, pf, ebx, @eax:32, edi`}, ce qui nous indique que tous ces objets dépendent de notre état de teinte initial.

L'analyse de teinte dynamique permet donc de suivre une ou plusieurs marques au cours d'une exécution. Dans notre exemple, la granularité de la teinte est de l'ordre des objets (registre et cellule mémoire) mais elle peut être de l'ordre du bit [97, 22].

3.2.2 Solveur de contraintes

Un problème de satisfiabilité modulo des théories (SMT) est est un problème de décision pour des formules de logique du premier ordre. Un solveur de contraintes (*SMT solver* [36, 25, 40, 20]) permet de savoir si une formule peut être satisfaite (on dit qu'elle est `SAT`). Si la formule est satisfaisable, le *SMT solver* est capable de fournir une interprétation (on parle alors de modèle) qui rend la formule vraie. Par exemple, la formule $x + y = 10$ est `SAT` et l'un des modèles la rendant vraie est $\langle x = 3, y = 7 \rangle$. À l'inverse, la formule $(a \vee b) - (a + b) + (a \wedge b) \neq 0$ est `UNSAT` car aucun modèle ne peut rendre la formule vraie. Un solveur de contrainte peut donc répondre `SAT` et fournir un modèle satisfaisant la contrainte, répondre `UNSAT` si aucun modèle ne satisfait la contrainte ou répondre `TIMEOUT` si le problème est indécidable (contrainte trop complexe).

3.2.3 L'exécution symbolique

Une exécution symbolique [61, 14] est une analyse qui permet de représenter les chemins d'un programme par un ensemble de contraintes, appelé *prédicats de chemin*, dont les contraintes sont liées aux entrées du programme. Cette méthode d'analyse est principalement utilisée afin de générer les entrées du programme permettant d'explorer ses chemins (voir Section 3.2.3.5).

```
int f(int x, int y, int z) {
  if (x == 0) {
    x = y + z;          /* pc1 */
  }
  else if (x == 1) {
    x = y - z;          /* pc2 */
  }
  else {
    x = -1;             /* pc3 */
  }
```

```
  return x;
}
```

Listing 3.1 – Exemple de code

Afin d'illustrer le principe d'une exécution symbolique, prenons l'exemple de code illustré par le Listing 3.1. La fonction f possède 3 chemins indiqués par les commentaires $pc1$, $pc2$ et $pc3$. Le rôle d'une exécution symbolique est d'associer à chacun de ces chemins un prédicat de chemin. Ces prédicats de chemin sont des formules logiques qui expriment les conditions nécessaires sur les variables de la fonction f. Dans notre exemple, les prédicats de chemin de la fonction f sont les suivants :

$$\phi_{pc1} \triangleq x_0 = 0 \wedge x_1 = y_0 + z_0$$
$$\phi_{pc2} \triangleq x_0 \neq 0 \wedge x_0 = 1 \wedge x_1 = y_0 - z_0$$
$$\phi_{pc3} \triangleq x_0 \neq 0 \wedge x_0 \neq 1 \wedge x_1 = -1$$

Les variables x_i, y_i et z_i sont des variables dites symboliques. Ces variables représentent les valeurs des variables aux différents points d'exécution.

Une fois les prédicats de chemins générés, il est possible de chercher les valeurs qui permettent de satisfaire un prédicat. Pour cela, on utilise un solveur de contraintes (voir Section 3.2.2). Un solveur de contraintes reçoit en entrée une formule logique C (ex. un prédicat de chemin) contenant des variables symboliques (ici x_0, x_1, y_0 et z_0) et renvoie soit un *modèle* respectant la formule C, soit *UNSAT* pour désigner qu'aucun modèle n'est possible, soit ne peut pas conclure car la formule est trop complexe (*TIMEOUT*).

3.2.3.1 L'exécution symbolique dynamique

Une exécution symbolique dynamique est une analyse symbolique dont la particularité est de construire le prédicat de chemin d'une exécution concrète [47, 49, 48, 51]. Cette construction peut se faire en parallèle d'une exécution concrète ou à *post-execution* depuis la récupération d'une trace d'exécution. Pour reprendre notre exemple du Listing 3.1, le Tableau 3.2 illustre la construction d'un prédicat de chemin en parallèle d'une exécution concrète avec pour valeurs concrètes initiales $x = 1$, $y = 10$, $z = 20$. Dans cet illustration, l'exécution concrète suit le chemin $pc2$ et illustre la construction du prédicat de chemin ϕ_{pc2} au cours de l'exécution.

Code source	Valeurs concrètes	Construction du prédicat de chemin
int f(int x, int y, int z) {	$\{x = 1, y = 10, z = 20\}$	
if (x == 0) {	$\{x = 1, y = 10, z = 20\}$	$\phi_{pc2^1} \triangleq (x_0 \neq 0)$
x = y + z ;	Non exécuté	
}		
else if (x == 1) {	$\{x = 1, y = 10, z = 20\}$	$\phi_{pc2^2} \triangleq (x_0 \neq 0) \wedge (x_0 = 1)$
x = y - z ;	$\{x = -10, y = 10, z = 20\}$	$\phi_{pc2^3} \triangleq (x_0 \neq 0) \wedge (x_0 = 1) \wedge x_1 = y_0 - z_0$
}		
else {	Non exécuté	
x = -1 ;	Non exécuté	
}		
return x ;	$\{x = -10, y = 10, z = 20\}$	$\phi_{pc2^4} \triangleq (x_0 \neq 0) \wedge (x_0 = 1) \wedge x_1 = y_0 - z_0$
}		

TABLE 3.2 – Construction du prédicat de chemin ϕ_{pc2} suivant une exécution concrète

3.2.3.2 Notion de concrétisation

L'avantage qu'offre une DSE est qu'elle peut s'appuyer sur un état concret pour alléger [1] la taille des expressions symboliques lors de la construction de son prédicat de chemin, on parle alors de concrétisation. La concrétisation formalisée dans [47, 34] est le fait de remplacer une expression symbolique à un point de contrôle par une valeur concrète. Cette valeur peut être issue de l'état concret de l'exécution au même point de contrôle ou alors générée de façon arbitraire.

	Valeurs concrètes	Prédicat de chemin
ex.1	$\{x = -10, y = 10, z = 20\}$	$\phi_{pc2^4} \triangleq (x_0 \neq 0) \wedge (x_0 = 1) \wedge x_1 = 10 - z_0$
ex.2	$\{x = -10, y = 10, z = 20\}$	$\phi_{pc2^4} \triangleq (x_0 \neq 0) \wedge (x_0 = 1) \wedge x_1 = y_0 - z_0 \wedge y_0 = 10$

TABLE 3.3 – Exemple de concrétisation de la variable y sur le prédicat de chemin ϕ_{pc2^4}

Par exemple, imaginons que nous voulons concrétiser la variable y_0 du prédicat de chemin ϕ_{pc2^4}. La Tableau 3.3 illustre deux façons d'appliquer cette concrétisation. La première (ex.1) consiste à remplacer en lieu et place de la variable y_0 la valeur concrète désirée. La deuxième (ex.2) consiste à rajouter une contrainte au prédicat de chemin afin de contraindre la variable y_0 à la valeur concrète désirée. La façon

1. Quand on parle d'alléger ou de réduire la taille d'expressions symboliques, ou parle alors de réduction de complexité des expressions pour les solveurs SMT.

dont on applique une concrétisation peut avoir un impact sur la correction et la complétude du prédicat de chemin [34].

Ce principe de concrétisation est utile pour réduire la taille des expressions du prédicat de chemin considérées comme non pertinentes par un analyste ou connues pour être complexes pour les solveurs de contraintes. Par exemple, le prédicat de chemin d'une fonction d'allocation telle que `malloc` peut ne pas être intéressante pour l'analyste. Il peut donc décider de concrétiser le calcul de l'adresse d'allocation afin d'alléger son prédicat de chemin au retour de la fonction `malloc`, et donc, d'aider le solveur de contraintes dans la résolution des contraintes. Voici un deuxième exemple, on sait que l'exécution du programme passe par une fonction cryptographique (ex. `sha1`) et savons que cette dernière n'est pas réversible (propriété cryptographique) et donc trop complexe pour le solveur de contraintes (ex. trouver z tel que $hash(z) = 1234$). Dans cet exemple, il est intéressant de concrétiser le retour de cette fonction dans le prédicat de chemin.

3.2.3.3 Notions de correction et de complétude

D'après la définition donnée par Patrice Godefroid [47], on note ϕ_w le prédicat de chemin d'une trace d'exécution w d'un programme p et on considère que le prédicat de chemin est *correct* si toutes les entrées satisfaisant ϕ_w suivent le chemin w. On considère que le prédicat de chemin est *complet* si toutes les entrées qui suivent la trace w vérifie le prédicat de chemin ϕ_w.

On considère également qu'un algorithme d'exploration de chemin est correct si l'ensemble des prédicats de chemin ϕ sont corrects et on dit que l'algorithme est complet si tous les chemins du programme p sont trouvés.

3.2.3.4 L'exploration de chemins

L'exécution symbolique dynamique est souvent utilisée pour effectuer une exploration de chemins. Le principe est d'effectuer une première exécution qui produit un prédicat de chemin. En se basant sur ce premier prédicat de chemin, nous inversons des conditions de branchement afin de générer des modèles permettant d'emprunter des nouvelles branches. Ces nouvelles branches sont ensuite exécutées ce qui produit de nouveaux prédicats de chemin et l'opération est répétée suivant le critère de couverture visé.

La Figure 3.1 illustre ce concept d'exploration de chemins. On note φ l'état

symbolique à chaque nœud, π les conditions de branchement et ϕ le prédicat de chemin. Dans un premier temps nous définissons une liste de prédicats à satisfaire (WL) et une liste de prédicats empruntés (DL). La première exécution est établie avec des entrées générées aléatoirement afin de fournir une première trace et produit un premier prédicat de chemin ($\phi_1 = (\varphi_1 \wedge \pi_1) \wedge (\varphi_2 \wedge \pi_2)$). Ce prédicat est placé dans la liste DL. En se basant sur le précédent prédicat de chemin récupéré (ici ϕ_1), on établi une liste de nouveaux prédicats de chemin n'étant ni dans DL ni dans WL, par exemple : $\phi_{new1} = (\varphi_1 \wedge \pi_1) \wedge (\varphi_2 \wedge \neg\pi_2)$ ainsi que $\phi_{new2} = (\varphi_1 \wedge \neg\pi_1)$. Les nouveaux prédicats de chemin établis sont ensuite placés dans WL pour pouvoir être explorés. L'opération est répétée tant qu'il y a des prédicats à satisfaire dans WL.

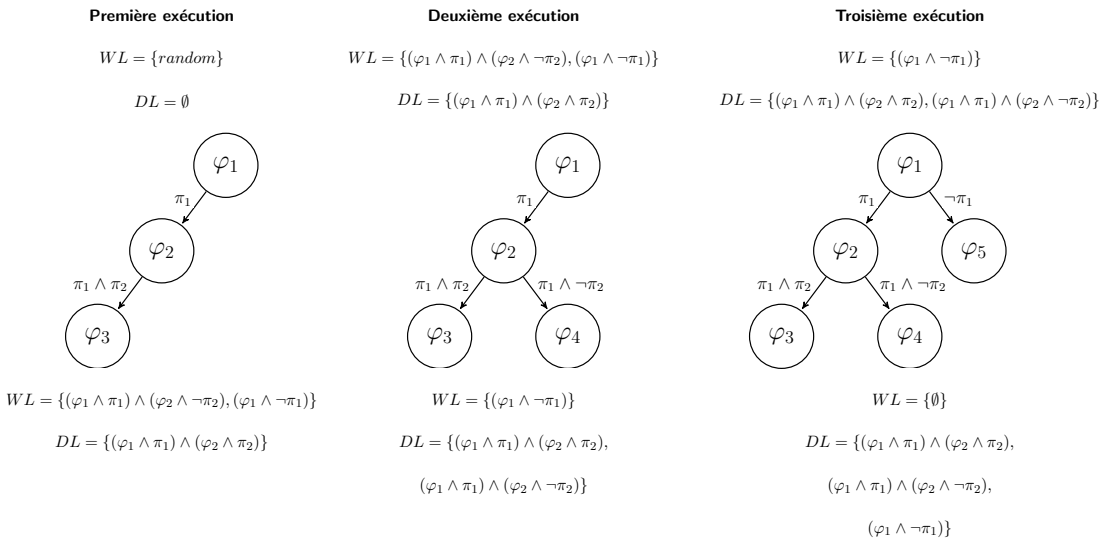

FIGURE 3.1 – Exemple d'exploration symbolique dynamique

3.2.3.5 Les stratégies de couverture des chemins

Les stratégies de couverture des chemins sont nombreuses [28, 27, 50], et à l'inverse du fuzzing, le DSE contrôle la manière dont est couvert le programme. Les stratégies les plus communes sont la *Breath-First-Search* (BFS) et la *First-Search* (DFS), toutes les deux implémentées dans DART [49] et SAGE [51].

On voit dans la littérature que certaines stratégies consistent à prendre des chemins aléatoires. C'est le cas de la stratégie *random-path* proposée par Klee [26]. Klee implémente également *cov-new*, une stratégie de probabilité des chemins basée

sur leur longueur, leur arité, le nombre de fois qu'ils ont été explorés et leur distance aux instructions non couvertes les plus proches. Il en choisit un au hasard pour continuer l'exécution, sachant que les chemins peu explorés sont favorisés.

Des approches plus dirigées ne visent pas une couverture maximale mais plutôt d'atteindre des emplacements précis dans un programme. Les travaux de Hicks [66] proposent une solution se basant sur un algorithme de plus court chemin dans le CFG appelé *shortest-distance symbolic execution* (SDSE). Les chemins ayant la plus courte distance à l'emplacement cible sont favorisés de cette manière.

On trouve également des stratégies dites *hybrides* [67, 90, 100] qui combinent fuzzing classique et exécution symbolique. L'exploration de chemins commence par injecter des valeurs aléatoires au programme afin de découvrir un maximum de chemins dans un temps très court (fuzzing classique). Dès que l'exploration de chemins est définie comme étant *bloquée* (aucun nouveau chemin n'est découvert), elle applique une exécution symbolique afin de résoudre les contraintes de branchement permettant d'amener à des chemins non couverts.

Comme nous pouvons l'imaginer il existe de nombreuse stratégies d'exploration dont des résumés sont proposés dans [86, 56, 65].

3.2.3.6 Autres optimisations

L'exécution symbolique est souvent critiquée en raison de la complexité à résoudre certaines contraintes ainsi que la complexité combinatoire qu'implique l'exploration des chemins. C'est pourquoi beaucoup de travaux portent sur la mise en place d'optimisations permettant de réduire ces complexités.

Optimisations sur les expressions : Klee [26] et EXE [28] mettent en place plusieurs optimisations telles que *Expression Rewriting* qui consiste à transformer des formes d'expressions connues par leur équivalence allant des simples transformations (ex. $x + 0 = x$), à des simplifications permettant de réduire la complexité de l'expression lors de son traitement par le solveur de contraintes (ex. $x * 2^n = x << n$) ou encore des simplifications linéaires (ex. $2 * x - x = x$). Klee [26] met également en place une optimisation appelée *Constraint Set Simplification* qui consiste à simplifier le prédicat de chemin au fur et à mesure qu'une contrainte y est rajoutée. Naturellement les contraintes sur les mêmes variables ont tendance à devenir plus spécifiques au cours de l'exécution. Par exemple, quand une contrainte telle que $x < 10$ est ajoutée au prédicat de chemin, suivie ultérieurement

d'une contrainte d'égalité telle que $x = 5$, Klee réécrit le prédicat de chemin en éliminant la première contrainte ajoutée car celle-ci peut être simplifiée par *true*. On peut également y trouver l'optimisation *Constraint Independence* qui consiste à déterminer les dépendances d'une requête dans un ensemble de contraintes. Par exemple, prenons l'ensemble de contraintes suivant : $\{i < j, j < 20, k > 0\}$. L'optimisation permet de savoir que pour une requête telle que $i = 20$, seules les deux premières contraintes de cet ensemble sont nécessaires. On peut également y trouver l'optimisation *Constraint Caching* qui consiste à mettre en cache les requêtes ainsi que leur résultat afin d'éviter, dans la mesure du possible, les appels au solveur de contraintes. Par exemple, le mécanisme de cache dans EXE [28] est déporté sur un serveur distant. Chaque entrée dans le serveur est sous la forme $\langle hash, result \rangle$ où $hash$ représente le hache $MD4$ de la représentation syntaxique de la requête et $result$ la réponse du solveur de contraintes (un modèle satisfaisant la contrainte, *unsat* ou *unknown*). Avant d'appeler le solveur de contraintes pour une requête q, EXE vérifie si la réponse est disponible en cache sur le serveur, si c'est le cas, le résultat est renvoyé depuis le mécanisme de cache. Dans le cas contraire, une requête est effectuée au solveur de contraintes et le résultat est mis en cache. Un système de cache au niveau des formules est facile à faire mais peu utile (rares sont les mêmes formules rejouées plusieurs fois), c'est pourquoi Klee [26] met en place un mécanisme de cache qui prend en compte les sous-formules ainsi que les sur-formules, ce qui demande une construction des prédicats de chemins et une représentation des formules dédiées.

Il existe également des optimisations pour les lectures et écritures mémoire tels que FAS (*Fast Array Simplification*) [43] appliquée en pré-processeur à une formule SMT. FAS propose une nouvelle représentation des tableaux sous la forme d'une liste de *maps* afin d'assurer le passage à l'échelle lors des lectures et écritures mémoire.

La thèse de Robin David [33] donne également un très bon aperçu de ces optimisations.

Optimisations sur les chemins : Les outils d'exploration de chemins ont montré leur efficacité à découvrir de nouveaux chemins, mais le problème auquel ils sont confrontés est de savoir comment gérer efficacement le nombre exponentiel de chemins découverts. Boonstoppel et al. [23] présentent une technique permettant de réduire le nombre de chemins à explorer en supprimant ceux qui doivent avoir des effets de bord identiques à certains chemins explorés précédemment.

Kuznetsov et al. [62] partent du principe qu'en fusionnant des états symbo-

liques issus de différents chemins, il est possible de réduire le nombre d'états à explorer dans un programme. À première vue, cela augmenterait la complexité des contraintes. Cependant, ils proposent deux nouvelles méthodes permettant de déterminer automatiquement quand et comment fusionner des états symboliques afin d'améliorer considérablement les performances de l'exécution symbolique. La première méthode *query count estimation*, qui permet d'estimer l'impact de chaque variable symbolique sur les performances du solveur de contraintes – les états sont fusionnés que si les performances promettent d'être avantageuses. Ainsi que la deuxième méthode *dynamic state merging*, permettant la fusion d'états symboliques en interagissant favorablement avec les stratégies de couverture de chemins.

Godefroid [46] introduit également les résumés symboliques de fonctions connues. Un résumé de fonction est une formule logique exprimée avec les pré-conditions des entrées et les post-conditions des sorties de la fonction. Le résumé est ensuite utilisé dès lors que la fonction ciblée est appelée ce qui évite de devoir construire sa représentation symbolique à chaque appel de cette dernière.

Lors de l'exploration de chemins, Bardin et al. [17] présentent une heuristique permettant d'éliminer autant que possible les chemins non pertinents. L'heuristique *Look-Ahead*, qui consiste à effectuer une analyse d'accessibilité (en termes d'éléments accessibles dans le CFG) pour décider si le chemin actuel doit être étendu ou non.

3.2.3.7 Le principal défi d'une DSE

Le principal défi d'une DSE est le passage à l'échelle et notamment deux aspects, (1) combattre l'explosion combinatoire qu'implique l'exploration des chemins, (2) trouver le juste milieu entre symbolisation et concrétisation.

Ces deux aspects sont fortement liés car la symbolisation et la concrétisation ont un impact sur l'exploration des chemins. La concrétisation permet de simplifier les expressions symboliques du prédicat de chemin et par conséquent offre un meilleur passage à l'échelle pour la résolution des contraintes mais implique généralement une perte de complétude lors de l'exploration des chemins. Prenons comme exemple le code illustré par le Listing 3.1. Imaginons que nous concrétisons la variable x par sa valeur concrète 0. Cette concrétisation nous fait perdre en complétude car nous ne pourrons pas trouver de modèle permettant d'explorer les chemins $pc2$ et $pc3$ de la fonction f.

Dans les sections qui suivent nous allons introduire Triton et discuter des choix

de conception du modèle symbolique concernant la correction et la complétude adoptées afin de répondre aux défis que nous nous sommes fixés.

3.3 Triton : Une API

Triton est un *framework* qui se présente sous la forme d'une bibliothèque développée dans l'optique de répondre aux défis présentés en Section 1.3. Triton est en libre accès [2] et offre des directives bas niveau permettant la mise en place d'analyses binaires par teinte, d'exécution symbolique dynamique et de représenter la sémantique opérationnelle d'un jeu d'instructions processeur (ISA) sous la forme d'arbres.

Triton vise à proposer des directives permettant la mise en place d'outils d'analyses plutôt qu'être un outil clef en main. Par expérience nous avons constaté qu'un seul outil est trop limité à l'utilisation de celui-ci dans un contexte bien précis et n'offre que très peu de latitude sur des cas d'usages très particuliers. Par exemple, en rétro-ingénierie les programmes à analyser sont souvent uniques et les objectifs très différents (voir Sections 2.1 et 2.2). C'est donc naturellement que le choix de conception de Triton s'est focalisé sur la mise en place d'une série de primitives dont la granularité d'analyse est celle d'une seule instruction et non d'un programme dans sa globalité. Ce niveau de granularité permet l'usage de Triton dans n'importe quel scénario et en combinaison avec n'importe quel autre outil permettant l'accès aux instructions assemblées (ex. GDB [89], BinaryNinja [7], IDA [54], Radare [74], Pin [78]...). Des exemples d'utilisation de Triton en scénario réel sont illustrés dans le Chapitre 4.

3.3.1 Architecture et fonctionnement

La Figure 3.2 illustre l'interaction des différents composants de la bibliothèque Triton. L'utilisateur commence par définir un contexte initial (ex. état des registres et de la mémoire). Si aucun contexte n'est donné, Triton initialise son contexte interne (mémoire et registres) par défaut avec les valeurs zéro. Ensuite, l'utilisateur fournit l'instruction à exécuter (sous la forme assemblée, ex : `48 b8 88 ...`). Une fois l'instruction donnée, Triton commence par désassembler cette dernière pour connaître sa mnémonique et ses opérandes (ici nous utilisons Capstone [77]

2. `https://triton.quarkslab.com`

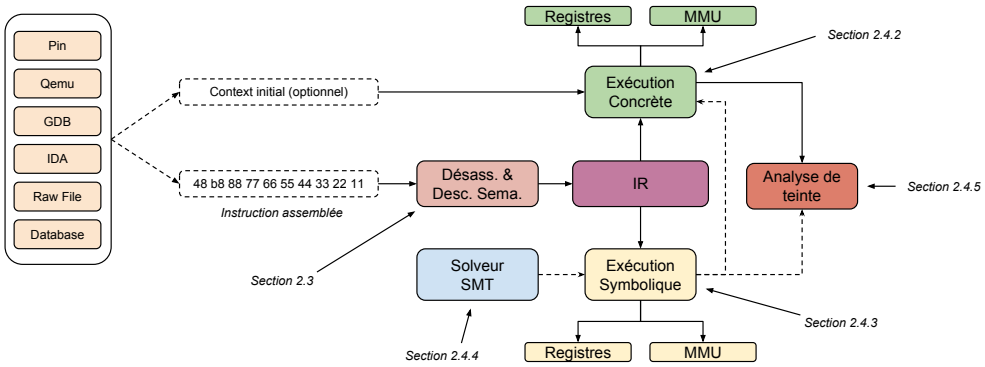

FIGURE 3.2 – Vue globale de la bibliothèque Triton

pour cet usage). Une fois le type de l'instruction connu, Triton représente l'instruction sous la forme d'arbres. Pour chacune des instructions nous avons décrit leur comportement sémantique sur les registres et la mémoire. Cette description suit les manuels techniques fournis par les concepteurs des CPUs (ex. " *Intel 64 and IA-32 Architectures Software Developer's Manual* " pour l'architecture `x86-64`). Actuellement, Triton supporte une partie des spécifications des architectures `i686`, `x86-64` et `AArch64` (ARM v8.5a).

Une fois les arbres de l'instruction courante construits, ils sont interprétés afin d'appliquer la sémantique de l'instruction sur les états concret (en vert dans la Figure 3.2), de teinte (en rouge) et symbolique (en jaune). Ces étapes d'interprétation et de calcul des états sont équivalents à une simulation de l'exécution d'une instruction et pour chaque exécution ces 3 étapes sont appliquées successivement. Le composant en charge de l'exécution symbolique a la possibilité d'accéder aux informations concrètes et de teinte afin d'alléger son prédicat de chemin dans le cas d'une application de politique de concrétisation (voir Section 3.5.3).

Quand vient l'envoi de la seconde instruction à exécuter, Triton prend comme états initiaux (concret, symbolique et de teinte) ceux de la précédente exécution et met à jour ses états internes en accord avec la sémantique de la nouvelle instruction exécutée et ainsi de suite, afin de simuler une trace d'exécution. Le fait que nos analyses reposent sur des arbres (qui est une forme de représentation intermédiaire) permet de garder générique nos analyses (évaluation concrète, symbolique et de teinte) sur les différentes architectures supportées (Intel et ARM). Dans le cas du rajout d'une architecture, seule l'encodage des instructions vers la représentation intermédiaire est nécessaire.

Afin d'illustrer la structure des arbres, prenons comme exemple l'instruction

`add rax, [rbx]` de l'architecture `x86-64`. Après exécution de l'instruction, le registre `rax` et le drapeau `ZF` seront affectés par l'AST de la Figure 3.3. Un AST sera également créé et affecté aux autres drapeaux : `AF`, `CF`, `OF`, `PF` et `SF`. Dans la Figure 3.3, la flèche en pointillés désigne une référence à un AST créé précédemment (partage d'arbre). L'algorithme commence par créer l'AST affecté au registre `rax` puis créer ensuite les AST des drapeaux en utilisant la référence de l'AST de l'expression `rax`.

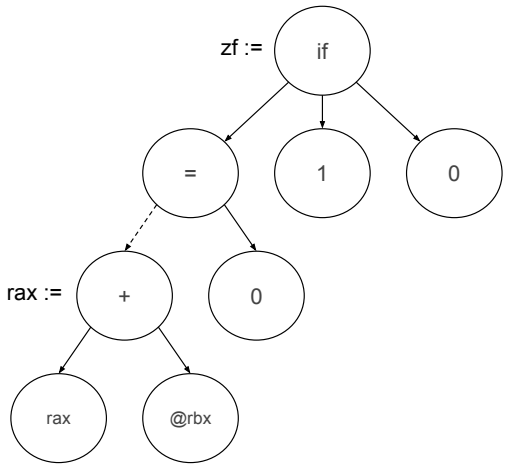

FIGURE 3.3 – AST affecté au registre `rax` et au drapeau `ZF` après exécution de l'instruction `add rax, [rbx]`

La construction de l'arbre s'effectue comme suit : les nœuds racines sont des opérateurs et les feuilles sont les opérandes. L'AST de la Figure 3.3 se lit comme suit : *On affecte 1 à ZF si le contenu de rax plus le contenu des cellules mémoire à l'adresse de rbx vaut 0 sinon on affecte 0 à ZF.*

La bibliothèque Triton fournit un accès à tous ses composants depuis une API accessible en C++ et en Python. Le choix du C++ a été adopté pour des aspects de performances et Python pour le côté " facile " d'intégration et d'utilisation dans une communauté *infosec* où la présence d'outils en Python est fortement établie.

3.3.2 Réponse aux défis

Pour conclure cette section, Triton est une bibliothèque d'analyse binaire dont la granularité d'analyse permet son intégration aisée avec différents outils (réponse

aux défis D1 et D2). Triton représente les instructions qu'il analyse sous la forme d'une représentation intermédiaire (réponse au défi D3). Triton interprète cette représentation intermédiaire d'une façon concrète et symbolique (réponse au défis D4) et y applique une analyse de teinte (réponse au défi D5).

Le modèle de Triton est décrit dans la Section 3.4, ses algorithmes d'analyse sont présentés en Section 3.5 et ses optimisations permettant le passage à l'échelle en Section 3.6).

3.4 Modèle de Triton

Lors de l'exécution d'une instruction, Triton décrit l'instruction sous la forme d'arbres et y applique successivement 3 algorithmes : un algorithme d'exécution concrète, puis un algorithme d'analyse de teinte et enfin un algorithme d'exécution symbolique dynamique. Dans les sections qui suivent nous allons décrire la structure de ces arbres et définir formellement ces algorithmes ainsi que les optimisations qui y sont appliquées.

3.4.1 Description syntaxique sous forme d'arbres

Quand Triton exécute une instruction, il commence par décrire l'instruction sous la forme d'arbres. Dans le modèle de Triton, une instruction peut produire plusieurs arbres (un arbre par destination) et en produit au minimum 1 (celui de l'affectation du pointeur d'instruction). Par exemple, l'instruction `add rax, rbx` produit 8 arbres affectés respectivement au : registre `rax` et aux drapeaux `AF`, `CF`, `OF`, `PF`, `SF` et `ZF` liés au calcul de l'addition. Puis un dernier arbre affecté au registre `rip` afin de déplacer le pointeur d'instruction vers la prochaine instruction à exécuter. Le Tableau 3.4 définit la grammaire de la description d'une instruction dans le modèle de Triton.

Note : *Quand nous parlons d'expressions, nous référençons le terme " expr " défini dans la grammaire illustré par le Tableau 3.4.*

Nous utilisons les opérateurs unaires ($\diamond_u$) et binaires ($\diamond_b$) classiques ($\neg$, $\vee$, $\wedge$, $+$, $-$, $\times$...) ainsi que les opérateurs sur les *bitvectors* dont la liste est en annexe dans le Tableau 5. L'opérateur @ désigne l'accès à une cellule mémoire sur 1 octet et $@_k$ un accès mémoire sur k octets. L'opérateur $\|$ désigne la concaténation des bits de

$inst$	$::=$	$\langle lhs := expr \rangle +$
lhs	$::=$	$registre \mid registre_{h..l} \mid @expr \mid @_k expr$
expr	$::=$	$(expr \ \diamond_b \ expr) \mid \diamond_u \ expr \mid @expr$
		$\mid @_k expr \mid (expr \ \| \ expr) \mid ite(expr, expr, expr)$
		$\mid extract(high, low, expr) \mid sx(n, expr) \mid zx(n, expr)$
		$\mid registre \mid constante$
$\diamond_u$	$::=$	Opérateurs unaires
$\diamond_b$	$::=$	Opérateurs binaires

TABLE 3.4 – Grammaire de la description d'une instruction dans le modèle de Triton

deux expressions (leurs tailles peuvent différer). L'opérateur $ite(c, e_1, e_2)$ désigne une expression conditionnelle. Si la condition c est vraie, alors l'opérateur évalue l'expression e_1, sinon e_2. L'opérateur $extract(high, low, expr)$ désigne l'extraction des bits allant de low à $high$ de l'expression $expr$ (également noté $expr_{h..l}$ pour faciliter la lisibilité). L'opérateur $sx(n, expr)$ désigne l'extension de n bits d'une expression signée (ex. $sx(3, 1001b) = 1111001b$) et l'opérateur $zx(n, expr)$ désigne l'extension de n bits d'une expression non signée (ex. $zx(3, 1001b) = 0001001b$). La sémantique opérationnelle appliquée à ces expressions est définie par l'algorithme d'exécution concrète décrit en Section 3.5.1.

3.4.2 Tailles des expressions

Nous travaillons au niveau *bitvectors* ce qui implique une taille fixe et connue des expressions de n bits. De plus les opérations mathématique susceptibles de créer un débordement $(+, \times, -, <<)$ sont en fait des opérations modulo 2^n. Les règles permettant de déterminer la taille d'une expressions sont décrites dans la Figure 3.4.

Les tailles peuvent aller de 1 à 512 bits, limite fixée arbitrairement et suffisante pour analyser les jeux d'instructions `i686`, `x86-64` et `AArch64`. Par exemple, 1 bit pour les drapeaux et 512 bits pour les registres les plus gros tel que `zmm0` (extension `AVX-512` sur `x86-64`). À noter également que lors d'une affectation, la source (*rhs*) doit avoir une taille identique à celle de la destination (*lhs*). Cela est également vrai pour la taille des opérandes e_1 et e_2 d'une expression binaire $e_1 \diamond_b e_2$. Les opérateurs sx et zx permettent de manipuler la taille des expressions en cas de besoin. Par exemple : si nous avons un opérateur binaire $e_1 \diamond_b e_2$ avec $size(e_1) \neq size(e_2)$, alors nous utiliserons l'opérateur d'extension zx pour ajuster la taille de l'opérande la

Tailles des $\mathbb{E}$xpressions

$$constante\ \frac{}{size(constante) = taille\ du\ bitvector} \qquad registre\ \frac{r \in Reg}{size(r) = taille\ du\ registre}$$

$$registre_{h..l}\ \frac{h \leq 512 \quad h \geq l \geq 0}{size(registre_{h..l}) = h - l + 1}$$

$$@e\ \frac{}{size(@e) = 8\ (bits)} \qquad @_k e\ \frac{k \geq 1 \quad k \leq 64}{size(@_k e) = k * 8\ (bits)}$$

$$e_1 \parallel e_2\ \frac{size(e_1) + size(e_2) \leq 512}{size(e_1 \parallel e_2) = size(e_1) + size(e_2)}$$

$$extract(h, l, e)\ \frac{h \leq 512 \quad h \geq l \geq 0}{size(extract(h, l, e)) = h - l + 1}$$

$$ite(c, e_1, e_2)\ \frac{size(e_1) = size(e_2)}{size(ite(c, e_1, e_2)) = size(e_1)}$$

$$sx(b, e)\ \frac{b \leq 512}{size(sx(b, e)) = size(e) + b} \qquad zx(b, e)\ \frac{b \leq 512}{size(zx(b, e)) = size(e) + b}$$

$$e_1 \diamond_b e_2\ \frac{size(e_1) = size(e_2)}{size(e_1 \diamond_b e_2) = size(e_1)} \qquad \diamond_u e\ \frac{}{size(\diamond_u e) = size(e)}$$

FIGURE 3.4 – Règles sur les tailles des expressions

plus petite et ainsi respecter les règles définies en Figure 3.4.

3.5 Algorithmes d'analyse

3.5.1 Algorithme d'exécution concrète (sémantique concrète)

Une fois l'instruction représentée sous la forme d'arbres, Triton commence par interpréter concrètement (émuler) le comportement de l'instruction. Cette section décrit l'algorithme d'exécution concrète.

3.5.1.1 Notations

On note Σ l'état concret à chaque point du programme. Cet état est décrit par une *map* qui lie chaque identifiant de registre (Reg) et adresse (entier) de cellule mémoire (Mem) à une valeur tel que $\Sigma : Reg \cup Mem \mapsto Val$. Dans le cas des registres, Val est un *bitvector* de la taille du registre en question. Pour les cellules mémoire, Val est un *bitvector* d'une taille de 8 bits. On note $\vdash_e$ l'évaluation d'une expression syntaxique de la grammaire du Tableau 3.4 en sa valeur entière ou booléenne. On note $\rightsquigarrow$ le passage d'un état concret Σ_n vers un état concret Σ_{n+1}.

On note $\Sigma[src]$ toute lecture dans la *map* d'état concret (ex. $\Sigma[addr]$ renvoie le contenu de la cellule mémoire à l'adresse $addr$). De même, on note $\Sigma[addr \leftarrow v]$ l'affectation de la valeur v dans la cellule mémoire à l'adresse $addr$. L'opérateur $@_k$ décrit une lecture de la mémoire sur k octets (*little* ou *big indian* en fonction de l'architecture). Dans les règles d'inférence qui suivent, la description est faite en *little indian*.

3.5.1.2 Évaluation des expressions (termes rhs)

Note : *Afin de faciliter la lisibilité et la compréhension du chapitre, nous plaçons en annexe les règles d'inférence considérées comme étant classiques.*

La sémantique opérationnelle des opérateurs arithmétiques sur les *bitvector* est présentée dans la Figure 21 placée en annexe. Par exemple, l'opérateur *bvand* effectue un `AND` bit à bit (à ne pas confondre avec le `AND` logique). L'opérateur $(e >>_u n)$ désigne un décalage non signé vers la droite de n bits de l'expression e

(ex. $1010b >>_u 2 = 0010b$). L'opérateur $>>_s$ désigne également un décalage de bits vers la droite mais sur une expression signée (ex. $1010b >>_s 2 = 1110b$). Les opérateurs $<<_u$ (non signé) et $<<_s$ (signé) désignent, quant à eux, des décalages de bits vers la gauche. La Figure 23 placée en annexe, quant à elle, définit la sémantique opérationnelle des opérateurs de comparaison ($=, \neq, \leq, \geq \ldots$) et des opérateurs booléens ($\wedge, \vee, \neg$).

$\mathbb{E}$xpressions

$$constante\ \frac{}{\Sigma, bv \vdash_e bv} \quad registre\ \frac{r \in Reg}{\Sigma, r \vdash_e \Sigma[r]} \quad @\ \frac{\Sigma, e \vdash_e addr}{\Sigma, @e \vdash_e \Sigma[addr]}$$

$$@_k\ \frac{\Sigma, e \vdash_e addr}{\Sigma, @_k e \vdash_e (\Sigma[addr + (k-1)] \parallel ... \parallel \Sigma[addr+1] \parallel \Sigma[addr+0])}$$

$$\parallel\ \frac{\Sigma, e_1 \vdash_e v_1 \quad \Sigma, e_2 \vdash_e v_2}{\Sigma, (e_1 \parallel e_2) \vdash_e (v_1 \parallel v_2)} \quad extract\ \frac{\Sigma, e \vdash_e v \quad h > l >= 0}{\Sigma, extract(h, l, e) \vdash_e v_{h..l}}$$

$$True - ite\ \frac{\Sigma, c \vdash_e v \quad v = True}{\Sigma, ite(c, e_1, e_2) \vdash_e e_1} \quad False - ite\ \frac{\Sigma, c \vdash_e v \quad v = False}{\Sigma, ite(c, e_1, e_2) \vdash_e e_2}$$

FIGURE 3.5 – Règles d'évaluation concrète des expressions

La Figure 3.5 décrit les règles d'évaluation des expressions arithmétiques pour les opérateurs de la grammaire du Tableau 3.4. Dans cette figure nous soulignons la présence d'opérateurs conditionnels (*ite*). L'algorithme d'évaluation de l'opérateur $ite(c, e_1, e_2)$ est le suivant : on commence par évaluer la condition booléenne c et si cette dernière est vraie ($True - ite$) alors on évalue l'expression e_1. Si l'évaluation de la condition c est fausse ($False - ite$), alors on évalue l'expression e_2.

Afin de faciliter certaines définitions (telle que la rotation de bits), nous avons introduit un ensemble d'opérateurs étendus (voir Figure 22 en annexe). L'opérateur *bvrol* décrit une rotation de bits vers la gauche. L'opérateur *bvror* décrit une rotation de bits vers la droite. L'opérateur *bvsmod* est le reste d'une division signée, le signe suivant le diviseur alors que l'opérateur *bvsrem* est le reste d'une division signée, le signe suit le dividende. L'opérateur *zx* étend la taille d'un *bitvector* avec des zéros (on note $0_{:s}$ un *bitvector* avec la valeur zéro d'une taille de s bits). L'opérateur $SIGNED - sx$ et $UNSIGNED - sx$ étendent la taille d'un *bitvector* en suivant le signe de ce dernier. Par exemple l'extension signée d'un *bitvector* non signé ($UNSIGNED - sx$) revient à faire une extension avec des zéros (zx).

3.5.1.3 Évaluation des instructions (termes lhs)

Dans les sections qui suivent quand on parle d'évaluation d'instruction on parle alors d'évaluation d'un terme lhs (Tableau 3.4). Les règles qui définissent cette évaluation sont décrites dans la Figure 3.6. Pour chaque terme lhs interprété, l'évaluation fait évoluer l'état concret Σ_n vers un état concret Σ_{n+1}. Ces règles sont classiques, on affecte à la destination, une source. Cette destination peut être un registre ($lhs - reg$), une partie d'un registre ($lhs - reg_{h..l}$), une cellule mémoire ($lhs - @$) ou un ensemble contigu de cellules mémoire ($lhs - @_k$).

$\mathbb{I}$nstructions

$$lhs - reg \ \frac{r \in Reg \quad \Sigma, src \vdash_e S \quad \Sigma_{new} \triangleq \Sigma[r \leftarrow S]}{\Sigma, r := src \rightsquigarrow \Sigma_{new}}$$

$$lhs - reg_{h..l} \ \frac{r \in Reg \quad \Sigma, src \vdash_e S \quad \Sigma_{new} \triangleq \Sigma[r \leftarrow (\Sigma[r]_{s-1..h+1} \parallel S \parallel \Sigma[r]_{l-1..0})]}{\Sigma, r_{h..l} := src \rightsquigarrow \Sigma_{new}}$$

$$lhs - @ \ \frac{\Sigma, dst \vdash_e D \quad \Sigma, src \vdash_e S \quad \Sigma_{new} \triangleq \Sigma[D \leftarrow S]}{\Sigma, @dst := src \rightsquigarrow \Sigma_{new}}$$

$$lhs - @_k \ \frac{\Sigma, dst \vdash_e D \quad \Sigma, src \vdash_e S \quad \Sigma_{new} \triangleq \Sigma[(\bigcup_{i=0}^{k-1} D + i \leftarrow S_{((i*8)+7)..(i*8)})]}{\Sigma, @_k dst := src \rightsquigarrow \Sigma_{new}}$$

FIGURE 3.6 – Règles d'évaluation concrète des instructions

3.5.1.4 Fonctionnement du flot de contrôle

Le flot de contrôle est implicite dans le modèle de Triton. Comme dit précédemment (en début de Section 3.4), la description sémantique d'une instruction processeur possède au minimum 1 arbre représentant le pointeur d'instruction (ex. `rip` sur `x86-64`). Cela signifie que nous avons au minimum un terme $reg := expr$ (règle $lhs - reg$ dans le Tableau 3.6) où reg représente le registre du pointeur d'instruction et $expr$ l'expression affectée à ce registre. Par conséquent, il possible de connaître la localisation d'une instruction processeur en accédant à l'état concret $\Sigma[r]$ où r est le registre du pointeur d'instruction. Les instructions conditionnelles sont représentées avec le terme $rip := ite(c, e_1, e_2)$ où le terme à droite de l'affec-

tation rhs est une expression conditionnelle (ite) et le terme de gauche lhs est le registre du pointeur d'instruction.

3.5.2 Algorithme d'analyse de teinte

Une fois l'état concret Σ mis à jour, Triton applique une analyse de teinte sur l'instruction. Dans Triton, la propagation de la teinte sur les registres et la mémoire a une granularité de l'ordre de taille des objets et n'est pas dépendante du flot de contrôle. Cette analyse a pour objectif de (1) aider l'analyste à suivre la propagation des données sur une trace d'exécution (2) guider le moteur symbolique dans les choix de conservation des expressions symboliques (voir Section 3.6.2).

3.5.2.1 Notations

On note Σ^t l'état de la teinte à chaque point du programme. Cet état est décrit par une *map* qui lie chaque identifiant de registre et adresse de cellule mémoire à une valeur booléenne tel que $Reg \mapsto bool \cup Mem \mapsto bool$. On note $\vdash_t$ l'évaluation d'une expression syntaxique de la grammaire (Tableau 3.4) en une valeur booléenne ($True$ égale teinté et $False$ égale non teinté). On note $\rightsquigarrow$ le passage d'un état de teinte Σ_n^t vers un état de teinte Σ_{n+1}^t. Par défaut, l'état initial de Σ^t ne contient aucune donnée teintée, c'est à l'utilisateur de définir (via l'API) quelles seront les données teintées initialement ou à des points spécifiques dans le programme.

3.5.2.2 Évaluation des expressions (termes rhs)

La Figure 3.7 illustre les règles de calcul de la teinte sur les expressions (termes rhs). Par exemple, l'évaluation ($\vdash_t$) d'un opérateur binaire ($e_1 \diamond_b e_2$) renvoie vraie si l'évaluation ($\Sigma^t, e_1 \vdash_t v_1 \quad \Sigma^t, e_2 \vdash_t v_2$) de l'une des deux opérandes ($v_1 \vee v_2$) est teintée (voir la règle *binaire* dans la Figure 3.7). Pour évaluer la teinte d'un accès mémoire (règle @), on commence par évaluer l'adresse concrète de l'accès mémoire ($\Sigma, e \vdash_e addr$) puis on regarde si cette cellule mémoire est définie comme étant teintée dans l'état de teinte ($\Sigma^t[addr]$).

$\mathbb{E}$xpressions

$$constante\ \frac{}{\Sigma, \Sigma^t, bv \vdash_t False} \quad registre\ \frac{r \in Reg}{\Sigma, \Sigma^t, r \vdash_t \Sigma^t[r]} \quad unaire\ \frac{\Sigma^t, e \vdash_t v}{\Sigma, \Sigma^t, \diamond_u e \vdash_t v}$$

$$binaire\ \frac{\Sigma^t, e_1 \vdash_t v_1 \quad \Sigma^t, e_2 \vdash_t v_2}{\Sigma, \Sigma^t, (e_1 \diamond_b e_2) \vdash_t (v_1 \vee v_2)} \quad @\ \frac{\Sigma, e \vdash_e addr}{\Sigma, \Sigma^t, @e \vdash_t \Sigma^t[addr]}$$

$$@_k\ \frac{\Sigma, e \vdash_e addr}{\Sigma, \Sigma^t, @_k e \vdash_t (\Sigma^t[addr + (k-1)] \vee ... \vee \Sigma^t[addr+1] \vee \Sigma^t[addr+0])}$$

$$True - ite\ \frac{\Sigma, c \vdash_e C \quad C = True \quad \Sigma^t, c \vdash_t vc \quad \Sigma^t, e1 \vdash_t v1}{\Sigma, \Sigma^t, ite(c, e_1, e_2) \vdash_t (vc \vee v1)}$$

$$False - ite\ \frac{\Sigma, c \vdash_e C \quad C = False \quad \Sigma^t, c \vdash_t vc \quad \Sigma^t, e2 \vdash_t v2}{\Sigma, \Sigma^t, ite(c, e_1, e_2) \vdash_t (vc \vee v2)}$$

FIGURE 3.7 – Règles d'évaluation de la teinte des expressions

3.5.2.3 Évaluation des instructions (termes lhs)

La Figure 3.8 illustre les règles de calcul de la teinte sur les instructions (termes lhs). Pour chaque terme lhs interprété, l'évaluation fait évoluer l'état de teinte Σ^t_n vers un état de teinte Σ^t_{n+1}. Ces règles sont classiques, on met à jour l'état de teinte lié à la destination en se basant sur l'état de teinte lié à la source. Cette destination peut être un registre ($lhs - reg$), une partie d'un registre ($lhs - reg_{h..l}$), une cellule mémoire ($lhs - @$) ou un ensemble de cellule mémoire ($lhs - @_k$). Tout comme pour l'évaluation des expressions, l'évaluation du terme $lhs - @$ commence par évaluer concrètement l'adresse de la destination ($\Sigma, e \vdash_e addr$), puis affecte la teinte de la source ($\Sigma^t, src \vdash_t t$) à l'adresse de la cellule mémoire ($\Sigma^t_{new} \triangleq \Sigma^t[addr \leftarrow t]$).

3.5.3 Algorithme d'exécution symbolique dynamique

Une fois l'état concret Σ et l'état de teinte Σ^t mis à jour. Triton interprète symboliquement l'instruction. L'algorithme d'exécution symbolique dynamique est le dernier des algorithmes à être appliqué. Cela signifie que ce dernier peut avoir accès à l'état concret Σ et l'état de teinte Σ^t pour concrétiser certaines valeurs lors

$\mathbb{I}$nstructions

$$lhs - reg \ \frac{r \in Reg \quad \Sigma^t, src \vdash_t t \quad \Sigma^t_{new} \triangleq \Sigma^t[r \leftarrow t]}{\Sigma, \Sigma^t, r := src \rightsquigarrow \Sigma, \Sigma^t_{new}}$$

$$lhs - reg_{h..l} \ \frac{r \in Reg \quad \Sigma^t, src \vdash_t t \quad \Sigma^t_{new} \triangleq \Sigma^t[r \leftarrow t]}{\Sigma, \Sigma^t, r_{h..l} := src \rightsquigarrow \Sigma, \Sigma^t_{new}}$$

$$lhs - @ \ \frac{\Sigma, e \vdash_e addr \quad \Sigma^t, src \vdash_t t \quad \Sigma^t_{new} \triangleq \Sigma^t[addr \leftarrow t]}{\Sigma, \Sigma^t, @e := src \rightsquigarrow \Sigma, \Sigma^t_{new}}$$

$$lhs - @_k \ \frac{\Sigma, e \vdash_e addr \quad \Sigma^t, src \vdash_t t \quad \Sigma^t_{new} \triangleq \Sigma^t[(\bigcup_{i=0}^{k-1} addr + i \leftarrow t)]}{\Sigma, \Sigma^t, @_k e := src \rightsquigarrow \Sigma, \Sigma^t_{new}}$$

FIGURE 3.8 – Règles d'évaluation de la teinte des instructions

de la construction du prédicat de chemin.

3.5.3.1 Notations

On note Σ^* l'état symbolique qui lie chaque identifiant registre (Reg) et adresse (entier) de cellule mémoire (Mem) à une expression symbolique φ en tout point du programme tel que : $r \in Reg \mapsto \varphi$ et $m \in Mem \mapsto \varphi$. On note ϕ la formule logique qui définit le calcul du prédicat de chemin en tout point du programme. On note $\vdash_s$ l'évaluation d'une expression syntaxique de la grammaire du Tableau 3.4 en une expression symbolique φ. On note $\rightsquigarrow$ le passage d'un état symbolique Σ^*_n vers un état symbolique Σ^*_{n+1}.

3.5.3.2 Évaluation des expressions (termes rhs)

Quand on évalue symboliquement ($\vdash_s$) un terme rhs, l'évaluation renvoie une expression symbolique (φ). La Figure 3.9 illustre les règles d'évaluation des expressions symboliques. L'évaluation des termes rhs vers une représentation symbolique est classique à l'exception des accès mémoires. Dans le modèle symbolique de Triton, l'évaluation symbolique d'un accès mémoire (règle @) se déroule comme suit : on commence par concrétiser le calcule de l'adresse mémoire ($\Sigma, e \vdash_e addr$), puis

on renvoie l'expression symbolique (φ) affectée à l'adresse de cette cellule mémoire ($@e \vdash_s \Sigma^*[addr]$).

$\mathbb{E}$xpressions

$$constante\ \frac{}{\Sigma, \Sigma^*, bv \vdash_s bv} \quad registre\ \frac{r \in Reg}{\Sigma, \Sigma^*, r \vdash_s \Sigma^*[r]} \quad unaire\ \frac{\Sigma^*, e \vdash_s \varphi_e \quad \varphi \triangleq \diamond_u \varphi_e}{\Sigma, \Sigma^*, \diamond_u e \vdash_s \varphi}$$

$$binaire\ \frac{\Sigma^*, e_1 \vdash_s \varphi_1 \quad \Sigma^*, e_2 \vdash_s \varphi_2 \quad \varphi \triangleq \varphi_1 \diamond_b \varphi_2}{\Sigma, \Sigma^*, (e_1 \diamond_b e_2) \vdash_s \varphi} \quad @\ \frac{\Sigma, e \vdash_e addr}{\Sigma, \Sigma^*, @e \vdash_s \Sigma^*[addr]}$$

$$@_k\ \frac{\Sigma, e \vdash_e addr}{\Sigma, \Sigma^*, @_k e \vdash_s (\Sigma^*[addr + (k-1)] \parallel ... \parallel \Sigma^*[addr+1] \parallel \Sigma^*[addr+0])}$$

$$ite\ \frac{\Sigma^*, c \vdash_s \varphi_c \quad \Sigma^*, e1 \vdash_s \varphi_1 \quad \Sigma^*, e2 \vdash_s \varphi_2}{\Sigma, \Sigma^*, ite(c, e_1, e_2) \vdash_s ite(\varphi_c, \varphi_1, \varphi_2)}$$

FIGURE 3.9 – Règles d'évaluation symbolique des expressions

Une étude [34] montre que la symbolisation et la concrétisation des accès mémoire jouent un rôle important dans le temps de résolution des contraintes par les *SMT solvers*. C'est pourquoi nous avons fait le choix de concrétiser tous les accès mémoire (lecture et écriture) afin de faciliter la résolution des contraintes et de respecter les défis que nous nous sommes fixés pour l'analyse de gros programmes (voir Section 1.3). Cependant cette décision sur la politique de concrétisation des accès mémoire a des conséquences sur la complétude du modèle symbolique (voir Section 3.5.3.5). Par exemple le moteur symbolique Binsec [38] a fait le choix de garder symbolique tous les accès mémoire ce qui lui offre une meilleure complétude mais augmente la complexité des expressions symboliques et nécessite donc des optimisations dédiées [43].

3.5.3.3 Évaluation des instructions (termes lhs)

Pour chaque terme lhs interprété, l'évaluation fait évoluer l'état concret Σ_n^* vers un état concret Σ_{n+1}^*. On affecte à la destination une expression symbolique φ. Cette destination peut être un registre ($lhs - reg$), une partie d'un registre ($lhs - reg_{h..l}$), une cellule mémoire ($lhs - @$) ou un ensemble contigu de cellules

mémoire ($lhs - @_k$). L'évaluation symbolique des termes lhs est classique à l'exception des accès mémoire. Tout comme pour l'évaluation des expressions, Triton commence par concrétiser le calcul de l'adresse mémoire ($\Sigma, e \vdash_e addr$), puis affecte l'expression symbolique (φ) à l'adresse de cette cellule mémoire ($\Sigma^*[addr \leftarrow \varphi]$).

$\mathbb{I}$nstructions

$$lhs - reg \; \frac{r \in Reg \quad r \neq pc \quad \Sigma^*, src \vdash_s \varphi \quad \Sigma^*_{new} \triangleq \Sigma^*[r \leftarrow \varphi]}{\phi, \Sigma, \Sigma^*, r := src \rightsquigarrow \phi, \Sigma, \Sigma^*_{new}}$$

$$lhs - reg_{h..l} \; \frac{r \in Reg \quad r \neq pc \quad \Sigma^*, src \vdash_s \varphi \quad \Sigma^*_{new} \triangleq \Sigma^*[r \leftarrow (\Sigma^*[r]_{s-1..h+1} \parallel \varphi \parallel \Sigma^*[r]_{l-1..0})]}{\phi, \Sigma, \Sigma^*, r_{h..l} := src \rightsquigarrow \phi, \Sigma, \Sigma^*_{new}}$$

$$lhs - @ \; \frac{\Sigma^*, src \vdash_s \varphi \quad \Sigma, e \vdash_e addr \quad \Sigma^*_{new} \triangleq \Sigma^*[addr \leftarrow \varphi]}{\phi, \Sigma, \Sigma^*, @e := src \rightsquigarrow \phi, \Sigma, \Sigma^*_{new}}$$

$$lhs - @_k \; \frac{\Sigma^*, src \vdash_s \varphi \quad \Sigma, e \vdash_e addr \quad \Sigma^*_{new} \triangleq \Sigma^*[(\bigcup_{i=0}^{k-1} addr + i \leftarrow \varphi_{((i*8)+7)..(i*8)})]}{\phi, \Sigma, \Sigma^*, @_k e := src \rightsquigarrow \phi, \Sigma, \Sigma^*_{new}}$$

FIGURE 3.10 – Règles d'évaluation symbolique des instructions

Ce choix de conception a été pris pour avoir une exécution symbolique qui passe à l'échelle même si elle ne couvre pas toutes les branches car nous utilisons principalement celle-ci pour de la recherche de vulnérabilités et non pour établir leur absence. Cependant, nous sommes conscients que cela peut être un frein à l'usage de Triton pour certaines missions telle que la vérification logicielle et avons la volonté de rajouter ce support dans un futur proche. Par exemple l'analyste aura la possibilité de passer d'une politique de concrétisation à une autre durant l'exécution comme décrit dans [34] afin de répondre au mieux à son besoin.

3.5.3.4 Constructions du prédicat de chemin

Il est possible d'obtenir le prédicat de chemin (ϕ) en appliquant la conjonction logique de tous les états symboliques (ex. $\phi \triangleq \Sigma^*_0 \wedge \Sigma^*_1 \wedge \Sigma^*_n$). Cependant, pour éviter que cela soit à la charge de l'utilisateur (et ainsi faciliter l'utilisation de bibliothèque), Triton propose d'obtenir le prédicat de chemin en tout point

de l'exécution. Pour cela, l'algorithme d'exécution symbolique introduit des règles spécifiques pour les instructions $lhs - reg$ et $lhs - reg_{h..l}$. Dans le cas de l'affectation d'une expression au registre de pointeur d'instruction, l'expression symbolique source est rajoutée dans le calcul du prédicat de chemin. La Figure 3.11 illustre la construction du prédicat du chemin pour l'instruction $lhs - reg$. Pour des raisons de lisibilité et pour faciliter la compréhension des règles, nous omettons volontairement l'instruction $lhs - reg_{h..l}$ dont le calcul du prédicat de chemin reste identique à celui de $lhs - reg$.

$\mathbb{I}$nstructions

$$lhs - reg \; \frac{r \in Reg \quad r = pc \quad \Sigma, src \vdash_e addr \quad \Sigma^*, src \vdash_s \varphi \quad \phi_{new} \triangleq \phi \wedge \varphi = addr \quad \Sigma^*_{new} \triangleq \Sigma^*[r \leftarrow \varphi]}{\phi, \Sigma, \Sigma^*, r := src \rightsquigarrow \phi_{new}, \Sigma, \Sigma^*_{new}}$$

FIGURE 3.11 – Règle pour la construction du prédicat de chemin

Le construction du prédicat de chemin fonctionne comme suit : Si le registre de destination est le pointeur d'instruction ($r \in Reg \quad r = pc$), on commence par évaluer concrètement l'adresse du saut (adresse de la prochaine instruction à exécuter, $src \vdash_e addr$). Ensuite, nous évaluons symboliquement ($\Sigma^*, src \vdash_s \varphi$) l'expression assignée au registre de pointeur d'instruction. Puis nous construisons une condition intégrant l'évaluation de l'adresse de destination ainsi que son expression symbolique que nous rajoutons au prédicat de chemin courant ($\phi_{new} \triangleq \phi \wedge \varphi = addr$). Ainsi, il est aisé pour un utilisateur d'obtenir le prédicat de chemin puis de développer un outil de couverture de chemins en inversant les conditions (ex. $\neg(\varphi = addr)$) et en utilisant un solveur de contraintes résoudre ces dernières.

3.5.3.5 Correction et complétude de l'exécution symbolique

Suivant la définition donnée en Section 3.2.3.3, l'exécution symbolique de Triton est correcte et complète si les indexes des cellules mémoire sont constants (non symbolique). En revanche, si l'exécution rencontre des accès mémoire dont leur indexation est symbolique, une concrétisation sera appliquée (voir les règles @ et $@_k$ des Figures 3.9 et 3.10) ce qui rend le prédicat de chemin incomplet dans tous les cas

et peut également être incorrect dans certains cas. L'impact de la concrétisation sur la complétude et la correction d'un prédicat de chemin est formalisé dans [47, 34].

Pour rendre le raisonnement symbolique de Triton correct et complet il faudrait (1) modifier notre façon d'appliquer la concrétisation (voir Section 3.2.3.2) afin de la rendre correcte [34], (2) modifier la logique appliquée à la mémoire afin de la rendre complète. Par exemple, au lieu de représenter la mémoire comme une *map* qui lie chaque adresse de cellule mémoire vers son expression ($m \in Mem \mapsto \varphi$), il faudrait représenter la mémoire (Mem) sous la forme d'un tableau symbolique tel que formalisé dans Binsec [38] en s'appuyant, par exemple, sur la logique `QF_ABV` de la SMT2-Lib.

Le rajout d'un modèle symbolique de la mémoire en s'appuyant sur la logique des tableaux ne serait pas sans impacter les performances du temps de résolution des contraintes. Toutefois, beaucoup d'efforts ont été faits sur la mise en place d'optimisations permettant de réduire cet impact [43, 75, 44].

3.6 Optimisations pour le passage à l'échelle

Il est commun de trouver des optimisations permettant de simplifier [3] les expressions symboliques afin d'aider les solveurs de contraintes à passer à l'échelle [28, 27, 35]. Dans cette section nous présentons deux optimisations permettant de réduire la taille des expressions symboliques durant la construction du prédicat de chemin. L'optimisation `ALIGNED_MEMORY` présentée dans la Section 3.6.1, permet de réduire la taille des expressions des accès mémoires et l'optimisation `ONLY_ON_TAINTED`, présentée dans la Section 3.6.2, permet de réduire la taille des expressions en se basant sur l'état de teinte. Ces deux optimisations peuvent se combiner et ont pour objectif d'aider la DSE à passer à l'échelle lors de l'analyse de gros binaires.

3.6.1 Réduction de la taille des arbres pour les accès mémoire

Certaines architectures telles que `i686` et `x86_64` peuvent accéder à la mémoire de façon non alignée contrairement à d'autres architectures comme `AArch64` où les accès mémoire doivent être alignés sur 4 octets. Par exemple en `x86_64` il est possible d'effectuer des lectures et des écritures en mémoire de 1, 2, 4, ou 8 octets sur toutes les adresses allouées. En revanche, sur `AArch64` seuls les accès mémoire

3. On parle souvent de réduire la taille des expressions symboliques.

de 4 octets ne sont autorisés et ils doivent être alignés sur une adresse de multiple de 4 (ex. `0x10000`, `0x10004`, `0x10008`, `0x1000c`...).

Une méthode facile et générique permettant de représenter la mémoire pour ces deux architectures est donc d'avoir une *map* qui lie chaque adresse mémoire à une expression φ de 8-bits ($\Sigma^*[Addr] \mapsto \varphi$).

Quand une expression de 32-bits est stockée en mémoire, elle est donc découpée en 4 expressions de 8-bits où chacune des ces expressions de 8-bits est affectée à une cellule mémoire (voir la règle $lhs - @_k$ dans la Figure 3.10). Dans le cas d'un chargement d'une expression de 32-bits depuis la mémoire, les 4 cellules mémoire sont concaténées pour former une seule expression de 32-bits (voir la règle $@_k$ de la Figure 3.9). La Figure 3.12 permet de visualiser ces deux règles lors d'une écriture et d'une lecture en mémoire.

Ces extractions et concaténations permettent une gestion plus facile des accès mémoire mais augmente la taille de la représentation intermédiaire, et à grande échelle, augmente donc considérablement la consommation mémoire. Afin d'illustrer cette augmentation de la taille de la représentation intermédiaire, prenons comme exemple le code du Listing 3.2. Cet échantillon stocke dans la mémoire la constante 1234 et la charge ensuite dans le registre `ebx`.

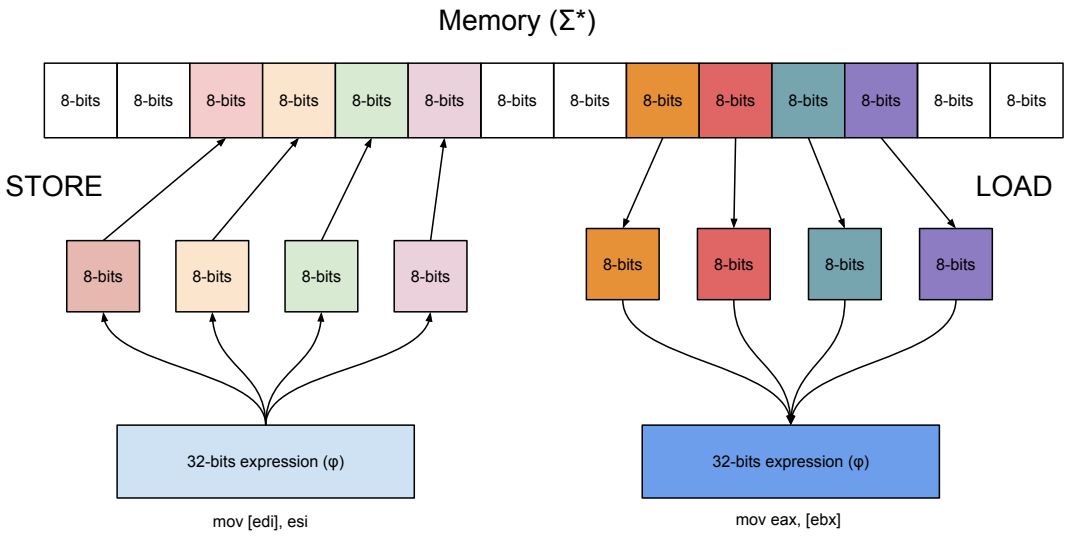

FIGURE 3.12 – Extraction et concaténation des expressions

```
1. mov eax, 1234
2. mov [edi], eax   # edi = 0x10000
3. mov ebx, [edi]   # edi = 0x10000
```

Listing 3.2 – Échantillon de code d'un STORE puis d'un LOAD.

La représentation associée au registre `ebx` est illustrée par la Figure 3.13. Cet arbre se lit comme suit : *le registre* **`ebx`** *est la concaténation (nœud vert) de 4 expressions (nœuds bleus) effectuant des extractions de 8-bit sur la constante 32-bits* **`1234`** *(nœuds jaunes)*. Les nœuds blancs étant les bits de poids fort et de poids faible pour les extractions.

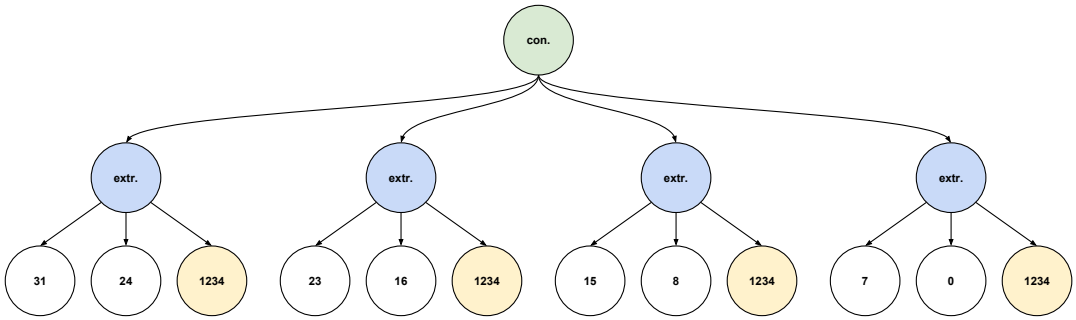

FIGURE 3.13 – AST d'`ebx` après exécution du Listing 3.2

Dans le cas d'une concaténation précédée d'extractions, si le résultat de la concaténation est identique à l'expression d'origine, l'arbre peut être réduit à l'expression d'origine. Dans l'exemple du Listing 3.2, l'expression symbolique du registre `ebx` peut donc tout simplement contenir la constante `1234` sur 32-bits, ce qui réduit la taille de l'arbre de 17 nœuds à 1 nœud.

Cette simplification est implémentée en utilisant un état de mémoire cache noté Σ^{mc}. Cet état est décrit par une *map* qui lie des couples $\langle adresse, taille \rangle$ représentant les accès mémoire à leurs expressions symboliques φ de la taille de leurs accès, tel que $\Sigma^{mc} : \langle adresse, taille \rangle \mapsto \varphi_{:taille}$.

Quand une affectation en mémoire a lieu ($STORE$), Σ^* est mis à jour comme illustré dans la Figure 3.10 et une entrée est également rajoutée dans l'état Σ^{mc} avec comme clef l'adresse et la taille de l'accès mémoire et comme contenu l'expression symbolique de la source (voir la règle $lhs - @_k$ dans la Figure 3.14). Lors de l'ajout d'une entrée dans l'état Σ^{mc} des vérifications sont faites pour supprimer les entrées qui se chevauchent (ex. $\langle 1000, 4 \rangle$ et $\langle 1002, 1 \rangle$). Lors de l'ajout de l'entrée $\langle 1002, 1 \rangle$, l'algorithme itère sur toute les entrées et supprime celles qui chevauche

la dernière entrée telle que $\langle 1000, 4\rangle$. Cette vérification est décidable et est omise dans la Figure 3.14 pour des raisons de lisibilité. Lors d'une opération de lecture de la mémoire ($LOAD$, règle $@_k$) une vérification est faite afin de voir si l'accès est présent dans l'état Σ^{mc}. Si l'accès est présent, l'expression source est récupérée depuis l'état Σ^{mc} (voir règle $@_k - T$). Dans le cas contraire, si l'accès n'est pas présent, l'expression est construite depuis l'état symbolique Σ^* (voir règle $@_k - F$).

$\mathbb{E}$xpression

$$@_k - T \ \frac{\Sigma, e \vdash_e addr \quad \langle addr, k\rangle \in \Sigma^{mc}}{\Sigma, \Sigma^*, @_k e \vdash_s \Sigma^{mc}[\langle addr, k\rangle]}$$

$$@_k - F \ \frac{\Sigma, e \vdash_e addr \quad \langle addr, k\rangle \notin \Sigma^{mc}}{\Sigma, \Sigma^*, @_k e \vdash_s (\Sigma^*[addr + (k-1)] \parallel ... \parallel \Sigma^*[addr+1] \parallel \Sigma^*[addr+0])}$$

$\mathbb{I}$nstruction

$$lhs - @_k \ \frac{\Sigma^*, src \vdash_s \varphi \quad \Sigma, e \vdash_e addr \quad \Sigma^{mc}_{new} \triangleq \Sigma^{mc}[\langle addr, k\rangle \leftarrow \varphi]}{\phi, \Sigma, \Sigma^{mc}, @_k e := src \rightsquigarrow \phi, \Sigma, \Sigma^{mc}_{new}}$$

FIGURE 3.14 – Règles pour les termes rhs et lhs avec l'optimisation `ALIGNED_MEMORY`

3.6.1.1 Expérimentations et résultats

Afin de mesurer l'efficacité de cette optimisation (`ALIGNED_MEMORY`) nous avons fait des expérimentations pour (1) mesurer le nombre de nœuds alloués pour les expressions symboliques durant l'exécution, (2) vérifier si cette optimisation a bien un impact sur la consommation mémoire compte tenu de la réduction d'allocation et conservation des nœuds, (3) évaluer si cette optimisation a un impact sur le temps d'exécution.

Pour mesurer le nombre de nœuds alloués pour les expressions symboliques, nous avons pris un ensemble de binaires et relevé toutes les 100 000 instructions le nombre de nœuds alloués et conservés dans Σ^* et Σ^{mc}. Plus nous conservons de nœuds dans les expressions symboliques plus la consommation mémoire augmente. La Figure 3.15 illustre la moyenne faite pour les différents binaires. La courbe en rouge est l'évolution de la conservation des nœuds sans aucune optimisation et on

peut constater que pour 2 000 000 d'instructions exécutées, nous conservons 4 087 947 nœuds dans Σ^* et Σ^{mc}. La courbe noire, quant à elle, représente l'évolution de la conservation des nœuds avec l'optimisation `ALIGNED_MEMORY`. On peut voir que pour 2 000 000 d'instructions exécutées, nous conservons désormais 638 863 nœuds dans Σ^* et Σ^{mc}, soit une réduction de 84% comparée à une exécution sans optimisation.

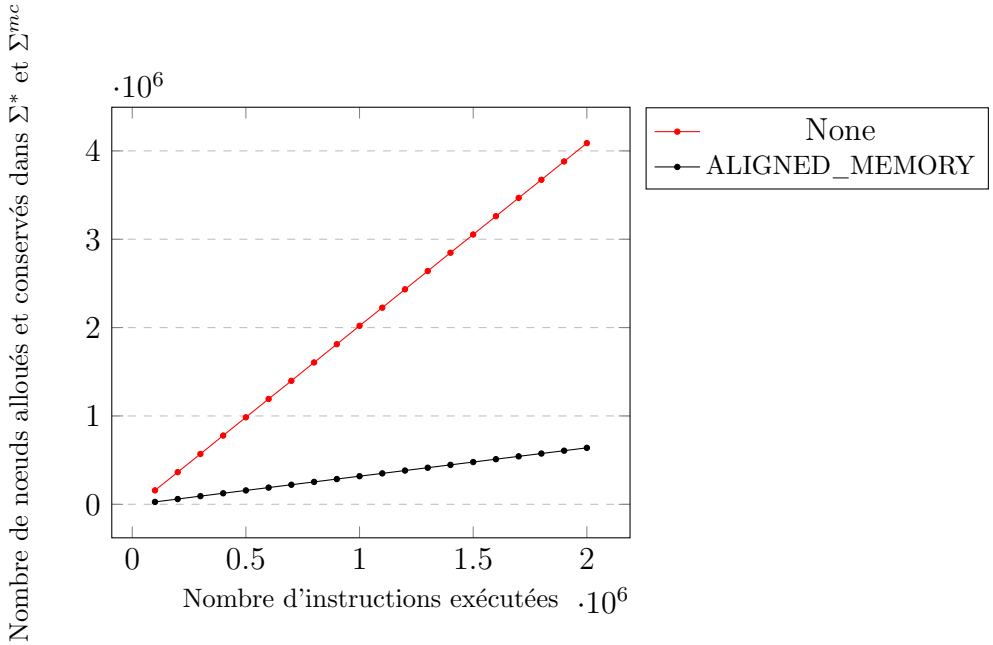

FIGURE 3.15 – Nombre d'allocation par instructions exécutées

Afin de vérifier si cette réduction des nœuds a bien un impact sur la consommation mémoire nous avons mesuré cette dernière toutes les 100 000 instructions exécutées. En moyenne pour les binaires considérés, la Figure 3.16 illustre l'évolution de la consommation mémoire lors des exécutions. La courbe en rouge indique l'évolution de la consommation mémoire sans optimisation et nous pouvons constater que pour 2 000 000 instructions exécutées nous consommons 4.1 gigaoctets de mémoire. La courbe en noire est l'évolution de la consommation mémoire avec l'optimisation `ALIGNED_MEMORY`. On peut voir que pour 2 000 000 instructions exécutées nous consommons 1.9 gigaoctets de mémoire soit une réduction de près de 52%.

Une question légitime maintenant est "*Est-ce que cette optimisation sacrifie le temps d'exécution pour gagner en consommation mémoire ?*". Pour vérifier si cette

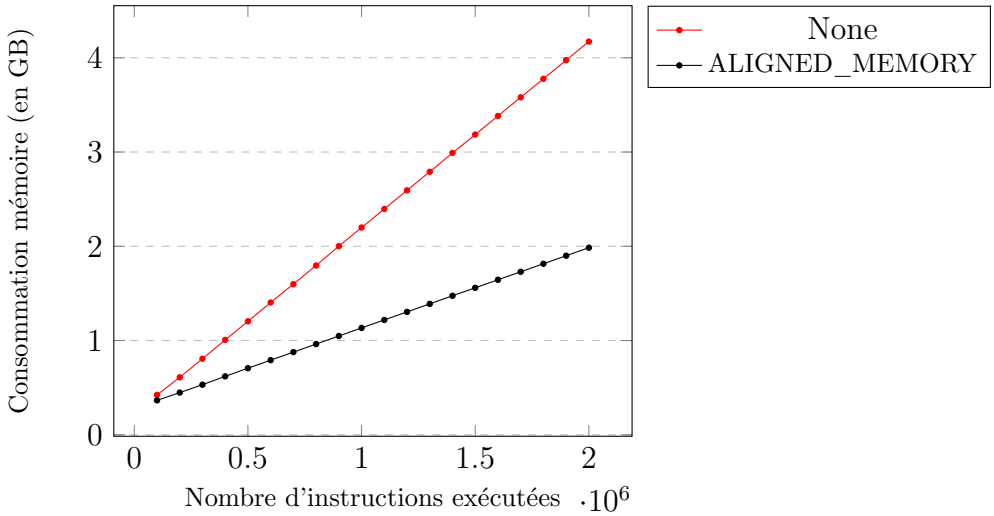

FIGURE 3.16 – Consommation mémoire par instructions exécutées

optimisation a un impact sur le temps d'exécution, nous avons relevé des points de mesure de temps (temps de traitement depuis le début de l'exécution) toutes les 100 000 instructions exécutées. La Figure 3.17 illustre le temps d'exécution moyen avec et sans optimisation. La courbe en rouge indique le temps d'exécution sans optimisation et nous pouvons constater que nous mettons 166 secondes pour exécuter 2 000 000 instructions. La courbe en noire est le temps d'exécution avec l'optimisation `ALIGNED_MEMORY`. On peut voir que nous mettons 161 secondes pour exécuter 2 000 000 instructions soit une variation légèrement inférieure à une exécution sans optimisation. Cette variation est vraiment minime et préférons conclure sur le fait que l'optimisation `ALIGNED_MEMORY` n'a pas d'impact sur le temps d'analyse.

Nous pouvons conclure que l'utilisation de l'optimisation `ALIGNED_MEMORY` permet en moyenne une réduction de près de 84% des nœuds conservés dans Σ^* et Σ^{mc}, et par conséquent, une réduction de près de 54% de la consommation mémoire sans pour autant impacter le temps d'exécution.

3.6.1.2 Correction et complétude de Σ^* avec l'optimisation

L'optimisation `ALIGNED_MEMORY` n'effectue aucune symbolisation ni aucune concrétisation. De plus, une expression symbolique créée avec `ALIGNED_MEMORY` est sémantiquement identique à une expression symbolique créée sans optimisation. Ainsi nous pouvons confirmer que l'utilisation de l'optimisation `ALIGNED_MEMORY`

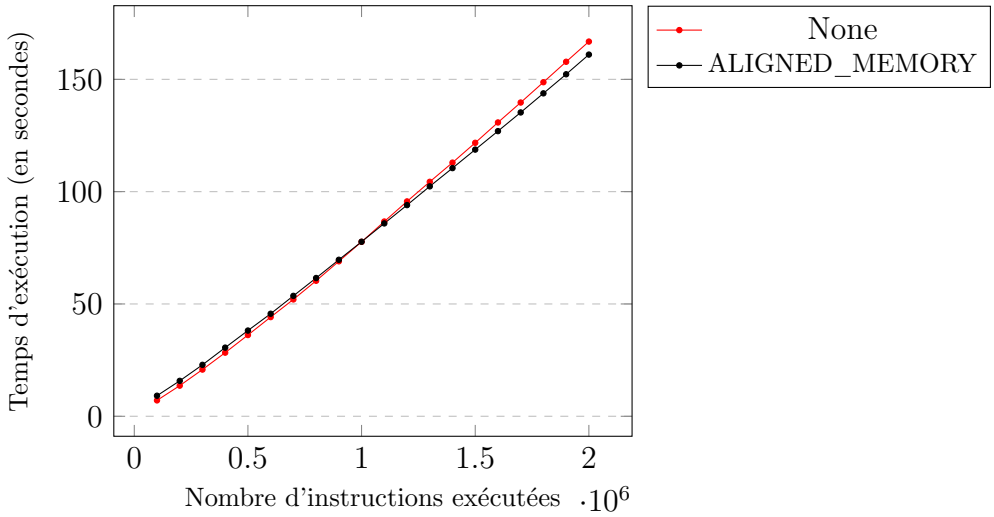

FIGURE 3.17 – Temps d'exécution par instructions exécutées

n'affecte pas la complétude ni la correction de l'exécution symbolique par rapport à une exécution sans optimisation.

3.6.2 Exécution symbolique guidée par la teinte

Triton propose un optimisation (`ONLY_ON_TAINTED`) permettant de garder symbolique toutes données teintées et de concrétiser tout ce qui ne l'est pas. Cela a pour objectif de réduire davantage la consommation mémoire (voir Section 3.6.1.1) en libérant (*free*) les allocations des expressions symboliques jugées *non pertinentes* par l'analyste (un exemple concret d'instructions non pertinentes est discuté dans la Section 5.2.1 lors de l'analyse de programmes obscurcis).

Lors de l'exécution symbolique, les termes *lhs* suivent les règles illustrées par la Figure 3.18 si l'optimisation `ONLY_ON_TAINTED` est activée. Ces règles sont un *raffinement* des règles de la Figure 3.10 en prenant en compte la teinte. Si la source est teintée nous affectons à la destination l'expression issue de l'état symbolique Σ^* (comme lors d'une exécution symbolique classique, Figure 3.10) et dans le cas contraire nous affectons à la destination la valeur courante de l'état concret Σ. Par exemple la règle $T - lhs - reg$ est l'affectation d'un terme teinté à un registre. La règle considère donc que la source est marquée comme teintée ($\Sigma^t, src \vdash_t True$) et l'évalue de façon symbolique ($\Sigma^*, src \vdash_s \varphi$) puis affecte l'expression symbolique au registre destination ($\Sigma^*_{new} \triangleq \Sigma^*[r \leftarrow \varphi]$). La règle $F - lhs - reg$, quant à elle, est l'affectation d'un terme non teinté à un registre. La règle considère donc que

la source est marquée comme non teintée ($\Sigma^t, src \vdash_t False$) et l'évalue de façon concrète ($\Sigma, src \vdash_e v$) puis affecte cette valeur au registre destination ($\Sigma^*_{new} \triangleq \Sigma^*[r \leftarrow v]$).

Instructions

$$T-lhs-reg\ \frac{r \in Reg \quad \Sigma^*, src \vdash_s \varphi \quad \Sigma^t, src \vdash_t True \quad \Sigma^*_{new} \triangleq \Sigma^*[r \leftarrow \varphi]}{\phi, \Sigma^*, r := src \rightsquigarrow \phi, \Sigma^*_{new}}$$

$$F-lhs-reg\ \frac{r \in Reg \quad \Sigma, src \vdash_e v \quad \Sigma^t, src \vdash_t False \quad \Sigma^*_{new} \triangleq \Sigma^*[r \leftarrow v]}{\phi, \Sigma^*, r := src \rightsquigarrow \phi, \Sigma^*_{new}}$$

$$T-lhs-reg_{h..l}\ \frac{r \in Reg \quad \Sigma^*, src \vdash_s \varphi \quad \Sigma^t, src \vdash_t True \quad \Sigma^*_{new} \triangleq \Sigma^*[r \leftarrow (\Sigma^*[r]_{s-1..h+1} \parallel \varphi \parallel \Sigma^*[r]_{l-1..0})]}{\phi, \Sigma^*, r_{h..l} := src \rightsquigarrow \phi, \Sigma^*_{new}}$$

$$F-lhs-reg_{h..l}\ \frac{r \in Reg \quad \Sigma, src \vdash_e v \quad \Sigma^t, src \vdash_t False \quad \Sigma^*_{new} \triangleq \Sigma^*[r \leftarrow (\Sigma[r]_{s-1..h+1} \parallel v \parallel \Sigma[r]_{l-1..0})]}{\phi, \Sigma^*, r_{h..l} := src \rightsquigarrow \phi, \Sigma^*_{new}}$$

$$T-lhs-@\ \frac{\Sigma, e \vdash_e addr \quad \Sigma^*, src \vdash_s \varphi \quad \Sigma^t, src \vdash_t True \quad \Sigma^*_{new} \triangleq \Sigma^*[addr \leftarrow \varphi]}{\phi, \Sigma^*, @e := src \rightsquigarrow \phi, \Sigma^*_{new}}$$

$$F-lhs-@\ \frac{\Sigma, src \vdash_e v \quad \Sigma, e \vdash_e addr \quad \Sigma^t, src \vdash_t False \quad \Sigma^*_{new} \triangleq \Sigma^*[addr \leftarrow v]}{\phi, \Sigma^*, @e := src \rightsquigarrow \phi, \Sigma^*_{new}}$$

FIGURE 3.18 – Règles d'évaluation symbolique des instructions guidée par la teinte

3.6.2.1 Expérimentations et résultats

L'efficacité de cette optimisation est fortement liée au contexte d'analyse et aux objectifs fixés par l'analyste.

D'une part, si l'analyste décide initialement de teindre toutes les données, cela revient à garder toutes les expressions symboliques et donc de ne pas bénéficier des propriétés de l'optimisation (aucune concrétisation ne sera faite). À grande échelle, si aucune donnée n'est teintée, cela consommera la mémoire jusqu'à saturation de cette dernière plus rapidement que si nous conservons que les expressions uniquement teintées.

D'autre part, si l'analyste ne teinte rien, cela revient à ne garder aucune expression symbolique, et par conséquent, il n'est pas possible de construire un prédicat de chemin complet (voir Section 3.6.2.2). La Figure 3.19 montre le nombre de nœuds alloués et utilisés dans Σ^* sans optimisation (courbe rouge), avec l'optimisation en teintant toutes les données (courbe bleue, ce qui revient à ne pas utiliser l'optimisation car toutes le expressions symboliques sont construites et conservées) et avec l'optimisation en ne teintant aucune donnée (courbe noire). Comme nous avons également pu le voir avec les expérimentations précédentes (Section 3.6.2.1) le nombre de nœuds alloués et utilisés dans Σ^* est directement lié à la consommation mémoire. C'est donc à l'analyste de teindre le minimum pour garder la complétude du prédicat de chemin la plus juste possible (en accord avec ses objectifs) et de laisser l'optimisation concrétiser le reste des expressions symboliques pour ralentir la consommation mémoire.

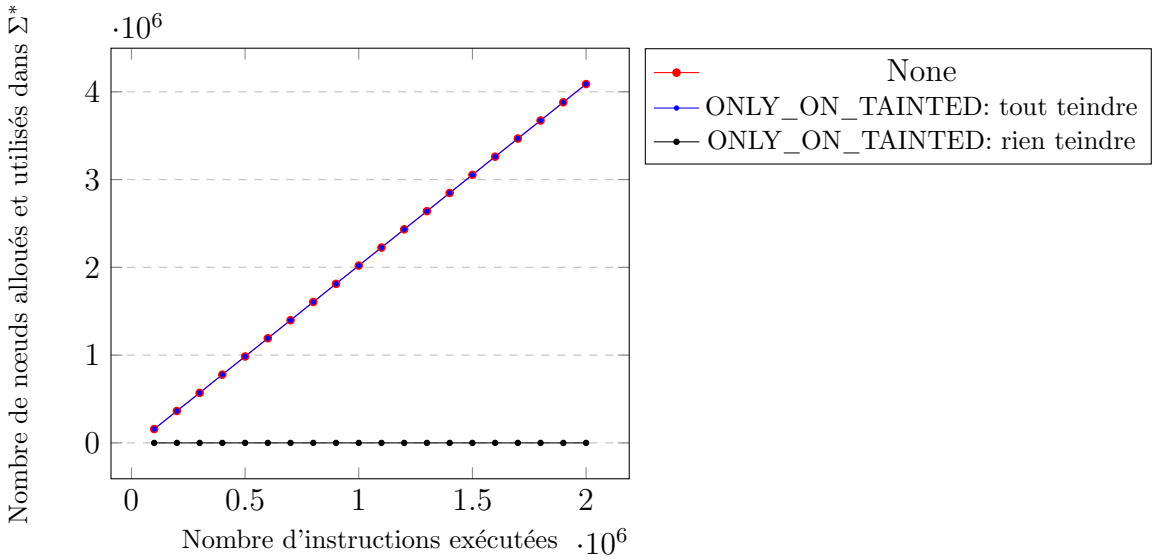

FIGURE 3.19 – Nombre d'allocation par instructions exécutées

Afin de mesurer l'impact de cette optimisation sur le temps d'analyse, nous avons relevé des points de mesure de temps (temps de traitement depuis le début de l'exécution) à toutes les 100 000 instructions exécutées. La Figure 3.20 illustre le temps d'exécution moyen avec et sans optimisation. La courbe rouge est le temps d'exécution sans optimisation, la courbe bleue est le temps d'exécution avec l'optimisation `ONLY_ON_TAINTED` en teintant toutes les données et la courbe noire est le temps d'exécution avec l'optimisation `ONLY_ON_TAINTED` en ne teintant aucune donnée.

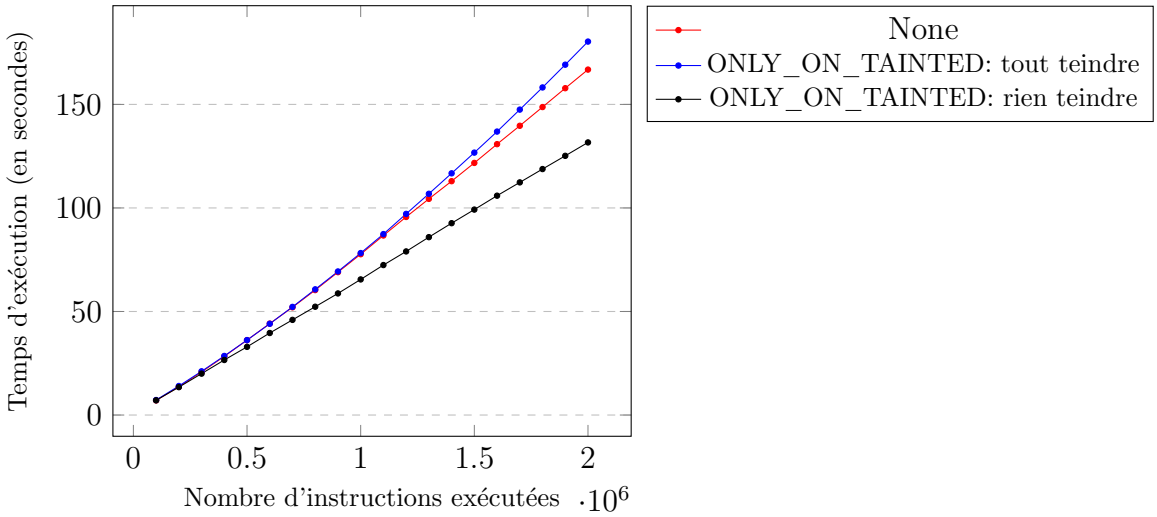

FIGURE 3.20 – Temps d'exécution par instructions exécutées

Compte tenu des résultats de nos expérimentations, nous pouvons conclure que l'optimisation `ONLY_ON_TAINTED` peut réellement avoir un impact fort sur la consommation mémoire. Cependant, de par l'usage de concrétisation, elle joue également un rôle important sur la complétude du prédicat de chemin (voir Section 3.6.2.2). L'optimisation `ONLY_ON_TAINTED` a également un impact sur le temps d'analyse, plus nous teintons de données, plus la teinte se répand durant l'exécution et le temps d'analyse est donc légèrement augmenté. À l'inverse, si aucune donnée n'est teintée, le temps d'analyse est diminué en raison de la non-construction des expressions symboliques.

3.6.2.2 Correction et complétude de Σ^* avec l'optimisation

L'optimisation `ONLY_ON_TAINTED` garde le même niveau de correction et de complétude si la teinte initiale est bien spécifiée. Notons que cela demande effectivement à l'analyste un travail supplémentaire. Si la teinte est sur-spécifiée, disons que toutes les données sont teintées, cette optimisation n'a aucun effet car elle ne concrétisera aucune expression. Si la teinte est sous-spécifiée, cela pourrait impacter la correction ainsi que la complétude du prédicat de chemin si les données non teintées sont utilisées dans le calcul de ce dernier.

3.7 Conclusion

Dans ce chapitre nous avons présenté le *framework* qui a été développé pour répondre aux défis que nous nous sommes fixés dans la Section 1.3. Nous avons décrit les algorithmes permettant d'émuler concrètement une trace d'exécution (Section 3.5.1), d'appliquer une analyse de teinte (Section 3.5.2) ainsi qu'une exécution symbolique (Section 3.5.3).

Afin de répondre au besoin d'analyser des gros programmes, nous avons mis en place deux optimisations permettant le passage à l'échelle d'une DSE. La première (`ALIGNED_MEMORY`), permet de réduire la taille des expressions symboliques des accès mémoires et la deuxième (`ONLY_ON_TAINTED`), permet de réduire la taille des expressions en se basant sur l'état de teinte. Ces deux optimisations peuvent se combiner, et comme le montre nos expérimentations, permettent de réduire drastiquement la consommation mémoire d'une DSE. Nous montrons dans le Chapitre 4 et 5 différentes utilisations pratiques de nos analyses.

3.8 Annexes

Terme Syntaxique	Type d'opérateur	Sémantique
and	boolean	$x \wedge y$
bvadd	bitvector	$x + y$
bvand	bitvector	$x \wedge y$
bvashr	bitvector	$x >>_s y$
bvlshr	bitvector	$x >>_u y$
bvmul	bitvector	$x \times y$
bvnand	bitvector	$\neg(x \wedge y)$
bvneg	bitvector	$-x$
bvnor	bitvector	$\neg(x \vee y)$
bvnot	bitvector	$\neg x$
bvor	bitvector	$x \vee y$
bvrol	bitvector	$((y << x) \vee (y >> (size - x)))$
bvror	bitvector	$((y >> x) \vee (y << (size - x)))$
bvsdiv	bitvector	$x \div_s y$
bvsge	boolean	$x \geq_s y$
bvsgt	boolean	$x >_s y$
bvshl	bitvector	$x << y$
bvsle	boolean	$x \leq_s y$
bvslt	boolean	$x <_s y$
bvsmod	bitvector	$(((x \mod y) + y) \mod y)$
bvsrem	bitvector	$(x - ((x \div_s y) \times y))$
bvsub	bitvector	$x - y$
bvudiv	bitvector	$x \div_u y$
bvuge	boolean	$x \geq_u y$
bvugt	boolean	$x >_u y$
bvule	boolean	$x \leq_u y$
bvult	boolean	$x <_u y$
bvurem	bitvector	$x \mod y$
bvxnor	bitvector	$\neg(x \oplus y)$
bvxor	bitvector	$x \oplus y$
distinct	boolean	$x \neq y$
equal	boolean	$x = y$
not	boolean	$\neg x$
or	boolean	$x \vee y$

TABLE 5 – Liste des opérateurs unaires et binaires

$\mathbb{E}$xpressions Arithmétiques (opérateurs classiques)

$$bvadd\ \frac{\Sigma, e_1 \vdash_e v_1 \quad \Sigma, e_2 \vdash_e v_2}{\Sigma, (e_1 + e_2) \vdash_e (v_1 + v_2)} \quad bvand\ \frac{\Sigma, e_1 \vdash_e v_1 \quad \Sigma, e_2 \vdash_e v_2}{\Sigma, (e_1 \ \wedge e_2) \vdash_e (v_1 \ \wedge v_2)} \quad bvashr\ \frac{\Sigma, e_1 \vdash_e v_1 \quad \Sigma, e_2 \vdash_e v_2}{\Sigma, (e_1 >>_s e_2) \vdash_e (v_1 >>_s v_2)}$$

$$bvlshr\ \frac{\Sigma, e_1 \vdash_e v_1 \quad \Sigma, e_2 \vdash_e v_2}{\Sigma, (e_1 >>_u e_2) \vdash_e (v_1) >>_u v_2} \quad bvmul\ \frac{\Sigma, e_1 \vdash_e v_1 \quad \Sigma, e_2 \vdash_e v_2}{\Sigma, (e_1 \times e_2) \vdash_e (v_1 \times v_2)} \quad bvnand\ \frac{\Sigma, e_1 \vdash_e v_1 \quad \Sigma, e_2 \vdash_e v_2}{\Sigma, \neg(e_1 \wedge e_2) \vdash_e \neg(v_1 \wedge v_2)}$$

$$bvneg\ \frac{\Sigma, e_1 \vdash_e v_1}{\Sigma, -e_1 \vdash_e -v_1} \quad bvnor\ \frac{\Sigma, e_1 \vdash_e v_1 \quad \Sigma, e_2 \vdash_e v_2}{\Sigma, \neg(e_1 \vee e_2) \vdash_e \neg(v_1 \vee v_2)} \quad bvnot\ \frac{\Sigma, e_1 \vdash_e v_1}{\Sigma, \neg e_1 \vdash_e \neg v_1}$$

$$bvor\ \frac{\Sigma, e_1 \vdash_e v_1 \quad \Sigma, e_2 \vdash_e v_2}{\Sigma, (e_1 \vee e_2) \vdash_e (v_1 \vee v_2)} \quad bvsdiv\ \frac{\Sigma, e_1 \vdash_e v_1 \quad \Sigma, e_2 \vdash_e v_2}{\Sigma, (e_1 \div_s e_2) \vdash_e (v_1 \div_s v_2)}$$

$$bvsub\ \frac{\Sigma, e_1 \vdash_e v_1 \quad \Sigma, e_2 \vdash_e v_2}{\Sigma, (e_1 - e_2) \vdash_e (v_1 - v_2)} \quad bvudiv\ \frac{\Sigma, e_1 \vdash_e v_1 \quad \Sigma, e_2 \vdash_e v_2}{\Sigma, (e_1 \div_u e_2) \vdash_e (v_1 \div_u v_2)} \quad bvurem\ \frac{\Sigma, e_1 \vdash_e v_1 \quad \Sigma, e_2 \vdash_e v_2}{\Sigma, (e_1 \bmod e_2) \vdash_e (v_1 \bmod v_2)}$$

$$bvxnor\ \frac{\Sigma, e_1 \vdash_e v_1 \quad \Sigma, e_2 \vdash_e v_2}{\Sigma, \neg(e_1 \oplus e_2) \vdash_e \neg(v_1 \oplus v_2)} \quad bvxor\ \frac{\Sigma, e_1 \vdash_e v_1 \quad \Sigma, e_2 \vdash_e v_2}{\Sigma, (e_1 \oplus e_2) \vdash_e (v_1 \oplus v_2)}$$

FIGURE 21 – Sémantique opérationnelle des expressions arithmétiques pour les opérateurs classiques sur les *bitvectors*

$\mathbb{E}$xpressions Arithmétiques (opérateurs étendus)

$$\begin{aligned}
bvrol(v_1, v_2, s) &\triangleq ((v_1 <<_u v_2) \vee (v_2 >>_u (s - v_1))) \\
bvror(v_1, v_2, s) &\triangleq ((v_1 >>_u v_2) \vee (v_2 <<_u (s - v_1))) \\
bvsmod(v_1, v_2) &\triangleq (((v_1 \bmod v_2) + v_2) \bmod v_2) \\
bvsrem(v_1, v_2) &\triangleq (v_1 - ((v_1 \div_s v_2) \times v_2)) \\
zx(s, v_1) &\triangleq (0_{:s} \parallel v_1) \\
SIGNED - sx(s, v_1) &\triangleq (((1 << s) - 1) \parallel v_1) \\
UNSIGNED - sx(s, v_1) &\triangleq zx(s, v_1)
\end{aligned}$$

FIGURE 22 – Définition des opérateurs étendus sur les *bitvector*

Expressions Booléennes (opérateurs classiques)

$$\mathit{and}\ \frac{\Sigma, e_1 \vdash_e v_1 \quad \Sigma, e_2 \vdash_e v_2}{\Sigma, (e_1 \wedge e_2) \vdash_e (v_1 \wedge v_2)} \quad \mathit{bvsge}\ \frac{\Sigma, e_1 \vdash_e v_1 \quad \Sigma, e_2 \vdash_e v_2}{\Sigma, (e_1 \geq_s e_2) \vdash_e (v_1 \geq_s v_2)} \quad \mathit{bvsgt}\ \frac{\Sigma, e_1 \vdash_e v_1 \quad \Sigma, e_2 \vdash_e v_2}{\Sigma, (e_1 >_s e_2) \vdash_e (v_1 >_s v_2)}$$

$$\mathit{bvsle}\ \frac{\Sigma, e_1 \vdash_e v_1 \quad \Sigma, e_2 \vdash_e v_2}{\Sigma, (e_1 \leq_s e_2) \vdash_e (v_1 \leq_s v_2)} \quad \mathit{bvslt}\ \frac{\Sigma, e_1 \vdash_e v_1 \quad \Sigma, e_2 \vdash_e v_2}{\Sigma, (e_1 <_s e_2) \vdash_e (v_1 <_s v_2)} \quad \mathit{bvuge}\ \frac{\Sigma, e_1 \vdash_e v_1 \quad \Sigma, e_2 \vdash_e v_2}{\Sigma, (e_1 \geq_u e_2) \vdash_e (v_1 \geq_u v_2)}$$

$$\mathit{bvugt}\ \frac{\Sigma, e_1 \vdash_e v_1 \quad \Sigma, e_2 \vdash_e v_2}{\Sigma, (e_1 >_u e_2) \vdash_e (v_1 >_u v_2)} \quad \mathit{bvule}\ \frac{\Sigma, e_1 \vdash_e v_1 \quad \Sigma, e_2 \vdash_e v_2}{\Sigma, (e_1 \leq_u e_2) \vdash_e (v_1 \leq_u v_2)} \quad \mathit{bvult}\ \frac{\Sigma, e_1 \vdash_e v_1 \quad \Sigma, e_2 \vdash_e v_2}{\Sigma, (e_1 <_u e_2) \vdash_e (v_1 <_u v_2)}$$

$$\mathit{distinct}\ \frac{\Sigma, e_1 \vdash_e v_1 \quad \Sigma, e_2 \vdash_e v_2}{\Sigma, (e_1 \neq e_2) \vdash_e (v_1 \neq v_2)} \quad \mathit{equal}\ \frac{\Sigma, e_1 \vdash_e v_1 \quad \Sigma, e_2 \vdash_e v_2}{\Sigma, (e_1 = e_2) \vdash_e (v_1 = v_2)}$$

$$\mathit{not}\ \frac{\Sigma, e_1 \vdash_e v_1}{\Sigma, \neg e_1 \vdash_e \neg v_1} \quad \mathit{or}\ \frac{\Sigma, e_1 \vdash_e v_1 \quad \Sigma, e_2 \vdash_e v_2}{\Sigma, (e_1 \vee e_2) \vdash_e (v_1 \vee v_2)}$$

FIGURE 23 – Sémantique opérationnelle des expressions booléennes

Chapitre 4

Cas pratique de Triton : Détection de prédicats opaques

Nous avons eu l'opportunité de participer à des missions d'audit de sécurité au sein de Quarkslab afin de tester Triton. Les objectifs de ce chapitre sont multiple :

1. Donner un exemple réel d'utilisation de Triton ;
2. Montrer que Triton peut se combiner avec d'autres outils (ex. IDA) ;
3. Montrer certaines fonctionnalités de Triton telles que les *hooks* [1] ;
4. Discuter sur les résultats et les limitations de Triton ;

Dans un premier temps, nous expliquons le contexte général de la mission ainsi que ses défis. Nous introduisons ensuite une approche permettant la détection de prédicats opaques dans les conditions de branchement. Puis nous effectuons des expérimentations afin de comparer notre approche avec les résultats d'une étude existante.

4.1 Contexte général et scénario

Lors d'une analyse statique de programme obscurci [85, 94], ce dernier contenait des prédicats opaques [71, 31, 70, 73] sur les conditions de branchement afin de

1. Les *hooks* permettent de modifier le contexte initial avant ou après l'application des règles d'analyses.

ralentir la rétro-ingénierie manuelle.

Un prédicat opaque (PO) est une formule renvoyant toujours la même valeur et est généralement utilisé dans le calcul d'une condition de branchement afin de tromper l'analyste en lui faisant croire que plusieurs chemins peuvent être empruntés. Par exemple, la Figure 4.1 illustre deux prédicats opaques toujours vrais et cela peu importe la valeur contenue dans a et b. Généralement a et b sont des données que peut contrôler l'utilisateur, ce qui va pousser l'analyste à croire qu'il peut emprunter plusieurs chemins.

$$\textbf{PO 1} \triangleq (a \times 2) \wedge 1 = 0$$
$$\textbf{PO 2} \triangleq (a \vee b) - (a + b) + (a \wedge b) = 0$$

FIGURE 4.1 – Exemple de Prédicats Opaques

Pour un analyste, repérer un prédicat opaque par une analyse manuelle (rétro-ingénierie statique) est une tâche fastidieuse, surtout si le calcul du prédicat opaque est éclaté à travers le programme ou si celui-ci est combiné avec des MBA (*Mixed Boolean-Arithmetics* [102, 41]).

$$\textbf{MBA 1} \triangleq (a \vee b) + (a \wedge b) \mapsto a + b$$
$$\textbf{MBA 2} \triangleq (a \wedge \neg b) \vee (\neg a \wedge b) \mapsto a \oplus b$$
$$\textbf{MBA 3} \triangleq (a \wedge b) * (a \vee b) + (a \wedge \neg b) * (\neg a \wedge b) \mapsto a * b$$
$$\textbf{MBA 4} \triangleq ((a \wedge \neg a) \wedge (\neg b \vee \neg a)) \wedge ((a \vee b) \vee (\neg b \vee b)) \mapsto a \oplus b$$

FIGURE 4.2 – Exemple de MBA

Les MBA sont des expressions combinant des opérateurs logiques ($\neg$, $\wedge$, $\vee$) et des opérateurs arithmétiques ($+$, $-$, $\times$, etc.) afin d'obscurcir une opération simple telle que $a + b$. La Figure 4.2 illustre plusieurs exemples de MBA. Par expérience, il est classique de trouver des MBA qui se combinent avec d'autres MBA afin de complexifier la compréhension de l'expression. Par exemple, le MBA 3 peut être combiné avec lui même ainsi qu'avec le MBA 1, et cela plusieurs fois. Noter que les MBA sont souvent utilisés dans le calcul d'un prédicat opaque. Par exemple, le Listing 4.1 est un prédicat opaque utilisant des MBA.

```
int foo(int a, int h) {
  /* ... */

  if (2 * (h & ~a) + (-1 * (~a | h) + (-1 * ~(a & h) + (2 * ~(a | h) + 1 * a)))) {
    /* ... */
  }
  else {
    /* ... */
  }

  /* ... */
}
```

Listing 4.1 – Exemple de prédicat opaque utilisant des MBA

En rajoutant plusieurs prédicats opaques dans une fonction, on augmente artificiellement le nombre de chemins que celle-ci possède, et par conséquent, on complexifie sa compréhension lors d'une analyse manuelle comme peut en témoigner le CFG de la Figure 4.3.

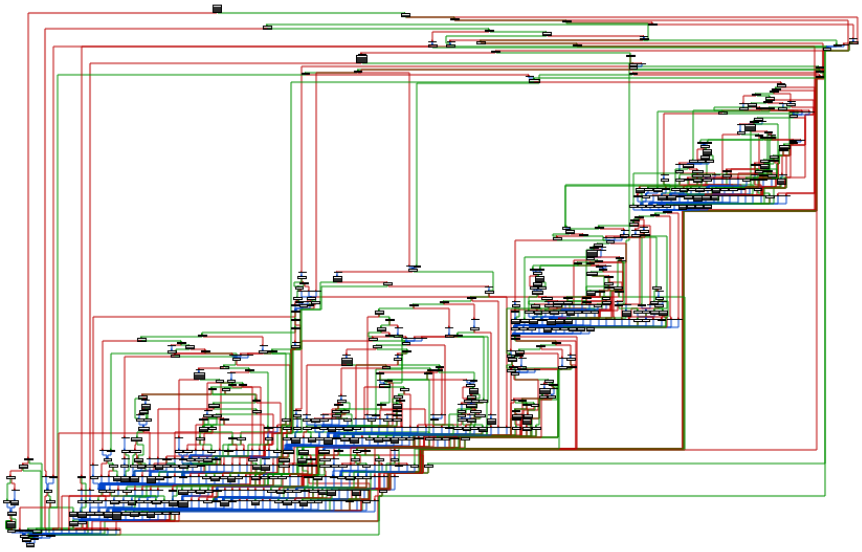

FIGURE 4.3 – Exemple de CFG d'une fonction obscurcie

Lors d'une analyse manuelle, il est possible de soupçonner qu'une condition est un prédicat opaque en regardant sa taille ainsi que la complexité de la formule. Cependant, ces hypothèse ne sont pas toujours viables (ex. code cryptographique), il est donc intéressant d'avoir des outils permettant à l'analyste de confirmer ces hypothèses. Il faut noter également que compte tenu du temps restreint lors des

missions d'analyse de programmes, il est important que l'analyste ait un maximum d'informations le plus rapidement possible pour comprendre au mieux le code qu'il audite. Pour cela, Triton peut aider à identifier rapidement la présence (vraie, fausse ou l'absence) de prédicats opaques dans les conditions de branchement en se reposant sur un raisonnement symbolique.

L'objectif de la mission était de comprendre et d'extraire un algorithme employé au sein d'une fonction obscurcie. Pour cela, il nous fallait dans un premier temps détecter et enlever les protections appliquées à la fonction. Le fait de détecter les protections appliquées, telles que les prédicats opaques, permet de se concentrer sur les portions de code en charge d'exécuter l'algorithme et d'ignorer les protions de code issues de la protection. Afin de détecter la présence de prédicats opaques au sein de la fonction obscurci nous avons donc utilisé Triton.

4.2 Détection symbolique de prédicats opaques

L'exécution symbolique est une méthode avérée pour la résolution des contraintes, elle peut donc se révéler intéressante pour la détection de prédicats opaques [85, 16, 70]. Un procédé permettant de détecter un prédicat opaque dans le calcul d'une condition de branchement, serait de vérifier si toutes les branches de chacune des conditions peuvent être accessibles. Si le raisonnement symbolique est correct et complet et que l'une des branches d'une condition X est inaccessible, cela peut signifier qu'il existe un prédicat opaque dans la condition X.

4.2.1 Prérequis

Triton étant un moteur d'analyse dynamique, nous sommes contraints de raisonner de façon dynamique. L'avantage que cela apporte c'est que nous traitons des expressions moins grosses qu'une analyse statique qui représenterait tout le flot de contrôle. L'inconvénient c'est que nous ne pouvons pas traiter l'ensemble des chemins de part l'explosion combinatoire que cela implique. C'est pourquoi nous avons fait le choix de faire porter notre analyse uniquement sur les blocs de base indépendamment des uns des autres. Il est donc nécessaire d'avoir au préalable l'ensemble des blocs de base de la fonction à analyser. Pour chacun de ces blocs de base, uniquement l'encodage binaire des instructions est nécessaire car ces derniers seront donnés à Triton qui s'occupera de les désassembler et de les exécuter symboliquement.

4.2.2 Méthodologie

La méthode mise en œuvre pour détecter des prédicats opaques dans les conditions de branchement est la suivante :

- ⋆ **Étape 0, récupération des blocs de base :** Désassemblage statique de code binaire puis construction d'un CFG pour obtenir la liste des blocs de base du programme. Cette étape peut être effectuée par des outils tels que IDA, Radare, Ghidra, ...
- ⋆ **Étape 1, construction des contraintes :** Pour chaque bloc de base terminant par un saut conditionnel, lancer une instance de Triton en rendant symbolique l'ensemble des registres ainsi que chaque lecture mémoire† – les écritures mémoire sont de fait inutiles. Cela correspond à suivre précisément les registres tout en abstrayant complètement la mémoire. Puis exécuter avec Triton chacune des instructions du bloc de base. Ce dernier se charge de construire les expressions symboliques de chaque affectation (cellules mémoire, registres et drapeaux) y compris celles du pointeur d'instructions. Si le bloc de base contient des appels externes, ignorer les appels et symboliser leur valeur de retour ce qui résulte en une *sur-approximation* de ces dernières.
- ⋆ **Étape 2, détection des prédicats opaques :** Pour chaque bloc de base, récupérer le prédicat de chemin ϕ construit par Triton et vérifier qu'il est `SAT`, puis faire de même avec sa négation $\neg\phi$. Si, l'un des deux prédicats ϕ ou $\neg\phi$ est `UNSAT`, cela signifie que l'une des deux branches est inatteignable et qu'il existe un prédicat opaque dans la condition de branchement. Attention, à l'inverse si les deux prédicats de chemin sont `SAT` cela ne garantit pas l'absence d'un prédicat opaque dans la condition de branchement car nous effectuons une sur-approximation du raisonnement symbolique (voir Section 4.2.3), tout étant symbolisé en début de bloc.

†**Note importante :** Par défaut Triton concrétise les accès mémoire et renvoie l'expression symbolique φ assignée à la cellule mémoire indexée (voir règle @ en Figure 3.9). Lors de l'étape 1, pour pouvoir renvoyer une nouvelle variable symbolique lorsqu'une lecture mémoire a lieu, il faut interagir sur le comportement initial du moteur symbolique de Triton. Pour cela, nous utilisons une fonctionnalité de Triton nous permettant de placer des *hooks* sur différentes étapes (lecture, écriture, ...) de ses composants internes (concret, symbolique, ...). Les *hooks* ne permettent pas de modifier le comportement des règles d'analyses mais uniquement de modifier le contexte avant et après l'application de ces

dernières. Nous utilisons donc le mécanisme de *hooks* pour initialiser les cellules mémoire avant chaque lecture de ces dernières (règle @) en y plaçant des nouvelles variables symboliques. Cela résulte en une sur-approximation des lectures mémoire. Un exemple d'utilisation des *hooks* est présentée ci-après.

Sur-approximation des lectures mémoire : Dans ce cas d'étude, l'usage de *hook* permettant de créer une nouvelle variable symbolique dès qu'une lecture mémoire à lieu, nous permet de prendre en compte l'ensemble des possibilités venant de la mémoire (peu importe l'indexation de la lecture). Pour illustrer ce concept, prenons comme exemples les pseudo-codes des listings suivants.

```
x := @a
y := @b

if (x > y) {
    ...
}

...
```

Listing 4.2 – Exemple 1

```
x := @a
y := @a

if (x == y) {
    ...
}

...
```

Listing 4.3 – Exemple 2

```
@a := x
y  := @a

if (x == y) {
    ...
}

...
```

Listing 4.4 – Exemple 3

Toutes les cellules mémoire lues se verront assigner une nouvelle variable symbolique. Dans l'exemple du Listing 4.2, au moment de la lecture mémoire $@a$, Triton concrétise l'adresse pointée par a (disons `0x1000`) et assigne son contenu à la variable x. Ici, avec l'usage des *hooks*, on va donc construire une variable symbolique (disons v_1) et l'assigner à la variable x ($x := v_1$). On fait de même avec la lecture $@b$ ($y := v_2$). Au moment de la condition de branchement, nous avons donc un prédicat avec deux variables symboliques ($v_1 > v_2$) et cela peu importe la valeur des adresses initiales.

Dans l'exemple du Listing 4.3 les variables x et y sont initialisés avec le contenu mémoire pointé par a. Dans cet exemple, x et y se verront assigner deux variables symboliques distinctes alors qu'elles pointaient sur la même adresse mémoire. Lors de la condition de branchement le prédicat sera alors $vrai$ ou $faux$ alors qu'il devrait toujours être $vrai$. Ce qui résulte en une sur-approximation des valeurs.

Dans l'exemple du Listing 4.4 la variable x est issue d'une assignation précédente (ex. bloc de base parent), on dit alors qu'elle est externe au périmètre local du bloc de base courant et donc symbolique ($x := v_1$). Lors de l'instruction $@a := x$, la

variable v_1 est placée à l'adresse de a mais lors de deuxième instruction, nous chargeons tout de même une nouvelle variable symbolique dans y ($y := v_2$) car une lecture mémoire a lieu et tout contenu lu se voit symbolisé. Lors de la condition de branchement le prédicat sera alors $vrai$ ou $faux$ alors qu'il devrait toujours être $vrai$. Ce qui résulte en une sur-approximation des valeurs.

Travaux similaires pour la détection de POs : David et al. [16] on travaillé sur des travaux similaires en effectuant une analyse symbolique dite en arrière bornée (*Backward-Bounded DSE*). Cela signifie qu'ils partent de chaque condition de branchement dans le programme à une position X et effectue une exécution symbolique de $X - N$ à X où N est le nombre fixe d'instructions à remonter dans le CFG (dans leur étude la délimitation est fixée à 16 instructions). Notre méthode est d'ailleurs proche de *BB-SE* [16] mais avec une borne dynamique (fin d'un bloc de base) au lieu d'une taille fixe d'instructions et nous calculons en avant (à partir de la limite jusqu'à la condition de branchement) plutôt que en arrière (de la condition de branchement vers la limite). Cependant, tout comme David et al. nous calculons le même objet pre^k (prédécesseurs en moins de k étapes). Pour résumer, les différences sont essentiellement :

1. Borne dynamique mais au sein d'un même bloc de base ;
2. Sur-approximation agressive de la mémoire ;

Par ailleurs, notre méthode est proche de celle de DoSE [93]. Tout comme DosE, nous effectuons une analyse symbolique bloc par bloc en avant bornée. Leur méthode est principalement utilisée pour déterminer l'équivalence de deux blocs de base (détection de prédicat opaque de type *Two-Way*). Cependant ils peuvent également détecter la présence de prédicats opaques dans les conditions de branchement (classe de prédicats opaques qu'ils ne considèrent pas).

4.2.3 Correction et complétude

La méthode est dite correcte si le prédicat opaque trouvé est un vrai prédicat opaque et complète si tous les prédicats opaques sont trouvés. Le fait d'analyser les blocs de base indépendamment les uns des autres permet le passage à l'échelle sur des gros binaires mais rend l'analyse correcte (relativement au CFG) et incomplète. Nos limites sont similaires à celles de David et al. [16].

Correcte relativement au CFG, car pour chaque bloc de base nous prenons

comme état initial l'ensemble des possibilités (registres et cellules mémoire sont symboliques) en entrée de ce dernier. Cela signifie que si l'analyse est positive à la présence d'un prédicat opaque dans la condition de branchement, nous sommes certains que le prédicat opaque existe. Cependant, la correction de notre méthode est également dépendante de la correction de l'outil de désassemblage (ici IDA) fournissant les blocs de base.

Incomplète, car nous ne prenons pas en compte le prédicat de chemin depuis l'exécution du programme. Cela ne nous garantit donc pas l'absence d'un prédicat opaque dans la condition de branchement car le calcul de ce dernier peut être éclaté à plusieurs endroits dans le programme (dépendance sur des valeurs précédemment calculées). L'analyse ne contient donc pas de faux positif (tests positifs à tort, un PO trouvé est un vrai PO) mais peut contenir des faux négatifs (tests négatifs à tort, il se peut que des POs ne soient pas détectés).

```
01. int foo(int a, int h) {
02.   int temp = 0;
03.
04.   /* Premier prédicat opaque toujours vrai */
05.   if (((a | h) - (a + h) + (a & h)) == 0) {
06.     /* Construction d'une partie du second prédicat opaque */
07.     temp = 2 * (h & ~a);
08.   }
09.
10.   /*
11.    * Second prédicat opaque.
12.    * Son calcul est éclaté dans le code de la fonction (voir la variable temp).
13.    */
14.   if (temp + (-1 * (~a | h) + (-1 * ~(a & h) + (2 * ~(a | h) + 1 * a)))) {
15.     /* ... */
16.   }
17.
18.   /* ... */
19. }
```

Listing 4.5 – Exemple de prédicat opaque dont le calcul est éclaté

Le Listing 4.5 illustre un prédicat opaque dont le calcul est éclaté à travers le code d'une fonction. Dans cet exemple, un premier prédicat opaque est utilisé (ligne 5) pour construire la première partie (ligne 7) d'un deuxième prédicat opaque (ligne 14). Pour pouvoir détecter la présence du deuxième prédicat opaque (ligne 14), il faut connaitre le calcul affecté à la variable `temp` (ligne 07) et donc prendre en compte l'ensemble des chemins. Dans cet exemple, le calcul du deuxième prédicat opaque est éclaté dans le périmètre local de la fonction `foo` mais on peut imaginer son calcul éclaté à plusieurs endroits dans le programme.

La complétude est également liée au fait de pouvoir désassembler des blocs et

des sauts du programme. Cela peut poser problèmes sur l'analyse de code obscurci ou polymorphe.

4.2.4 Conception et usage des *hooks* avec Triton

Les étapes 1 et 2 (Section 4.2.2) sont implémentées avec Triton au sein d'un *plugin* IDA (disponible en Annexe). L'usage d'IDA nous permet de bénéficier des ses fonctionnalités telles que le désassemblage et la construction de CFG (étape 0), IDA permet également d'avoir un rendu graphique des résultats de l'analyse. Par exemple, teindre d'une couleur les blocs de base contenant des prédicats opaques afin d'aider visuellement l'analyste à localiser ces derniers.

Usage des *hooks* : Bien que Triton concrétise le calcul d'indexation d'une cellule mémoire (voir les règles $@_k$ et $lhs - @$ en Figures 3.9 et 3.10), il est tout de même possible de configurer Triton via le système de *hook* pour construire une nouvelle variable symbolique dès qu'une lecture mémoire a lieu. Le Listing 4.6 est un extrait du *plugin* fourni en Annexe. À la ligne `13`, nous ajoutons un *hook* avant toute lecture mémoire. Dès qu'une lecture mémoire a lieu, notre fonction `hook_memory_read` sera appelée avec pour paramètre le contexte de Triton et la mémoire lue (`mem` est un objet représentant l'ensemble des cellules mémoire lues pour cet accès). Dans cette *callback*, nous itérons sur chacune des cellules mémoire (ligne `04`) et nous convertissons chacune de ces cellules en une nouvelle variable symbolique (ligne `06`). Cela a pour effet d'écraser l'ancienne expression symbolique φ assignée à la cellule mémoire en la remplaçant par une nouvelle variable symbolique. Triton commence par appliquer l'analyse concrète avant l'analyse symbolique (voir Section 3.5) et ce *hook* est placé lors de l'analyse concrète (`GET_CONCRETE_MEMORY_VALUE`). Cela a pour objectif de modifier l'état concret initial de Triton avant l'application de l'analyse symbolique. Dès que l'analyse symbolique a lieu, elle prend comme valeur initiale la nouvelle variable symbolique fraichement créée dans la *callback* en appliquant la règle @ décrite en Figure 3.9.

```
01. ...
02.
03. def hook_memory_read(ctx, mem):
04.     for i in range(mem.getSize()):
05.         memi = MemoryAccess(mem.getAddress()+i, CPUSIZE.BYTE)
06.         ctx.convertMemoryToSymbolicVariable(memi)
07.     return
08.
09. ...
10.
11. ctx = TritonContext()
```

```
12. ctx.setArchitecture(ARCH.X86_64)
13. ctx.addCallback(hook_memory_read, CALLBACK.GET_CONCRETE_MEMORY_VALUE)
14.
15. ...
```

Listing 4.6 – Exemple de hook avec Triton

4.3 Expérimentations

Nous avons expérimenté notre *plugin* IDA sur deux expérimentations. La première expérimentation (Section 4.3.1) porte sur un cas d'étude dont nous contrôlons les sources et où nous avons injecté des prédicats opaques. La deuxième expérimentation (Section 4.3.2) porte sur la détection de prédicats opaques dans le *malware* X-Tunnel et où nous comparons nos résultats avec ceux d'une étude publique [16].

4.3.1 Première expérimentation

Dans un premier temps, nous avons testé l'approche sur un binaire obscurci dont nous avions les sources et nous avons intégrer 15 prédicats opaques [2] dans une des fonctions du programme. Le CFG de cette fonction est illustré par la Figure 4.4.

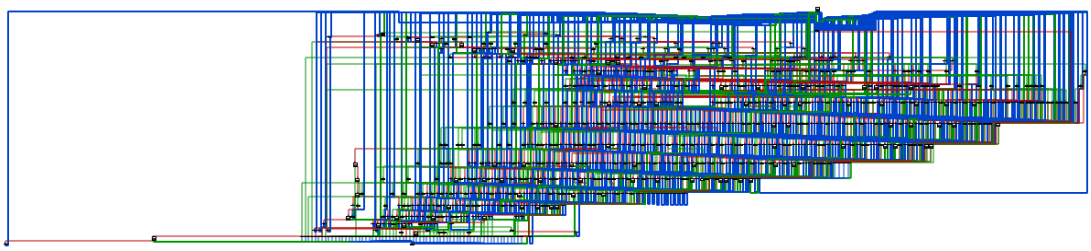

FIGURE 4.4 – CFG d'une fonction obscurcie contenant des prédicats opaques

Après avoir exécuté le *plugin* sur la fonction obscurcie, le résultat de ce dernier est affiché dans la console d'IDA (voir Figure 4.5). On peut voir que le *plugin* a

2. Merci à Alexandre Verine et Margaux Celle de nous avoir autorisés à utiliser leurs prédicats opaques.

analysé 927 blocs de base avec un total de 1429 branches testées (**étape 2** de l'analyse) et qu'il a détecté les 15 prédicats opaques injectés précédemment pour un temps total d'analyse de 84 secondes. Aucun faux positif n'a été détecté. Les prédicats opaques étant injectés à la main au préalable, les faux positifs et faux négatifs sont faciles à identifier.

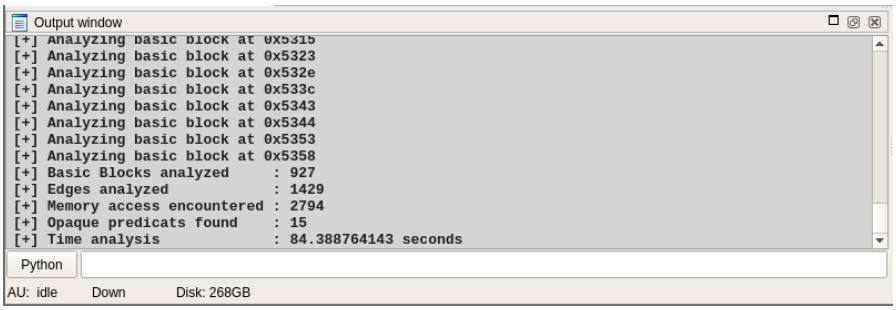

FIGURE 4.5 – Résultat dans la console d'IDA de notre *plugin* pour la recherche de prédicats opaques

Les prédicats opaques injectés possèdent deux variables (x et y) et mixent les opérateurs arithmétiques et booléens (MBA). La Figure 4.7 illustre des exemples type de ces derniers. Pour chaque prédicat opaque trouvé, le *plugin* nous informe si ce dernier est toujours vrai ou toujours faux (pour cela il suffit d'évaluer concrètement le prédicat opaque).

```
P01 = (-1) * ((~x) | y) + (-3) * (x ^ y) + (-1) * ((~y) | x) + (2) * (x | y) + (2) * (~(x & y))
P02 = (~(x | y)) + (x ^ y) + y + (-2) * 1 + ((~y) | x)
P03 = (2) * (~x) + (-1) * (x ^ y) + (-1) * ((~x) | y) + (-1) * (~(x | y)) + x
P04 = (-3) * (x ^ y) + (-1) * ((~x) | y) + (2) * (x | y) + (-1) * ((~y) | x) + (2) * (~(x & y))
P05 = (-1) * (~(x & y)) + (-2) * (y & (~x)) + (-1) * x + (2) * (x ^ y) + ((~x) | y)
P06 = (-1) * x + (-1) * (~(x & y)) + y + (-1) * (~(x | y)) + (2) * (~y)
P07 = (-1) * (x & (~y)) + (~x) + (-1) * ((~x) | y) + (-1) * (y & (~x)) + (x | y)
P08 = (2) * (x & (~y)) + (-1) * (x | y) + (-1) * (~(x ^ y)) + (-1) * (~(x & y)) + (2) * ((~x) | y)
P09 = ((~y) | x) + (-1) * y + (-1) * (x & (~y)) + (-1) * (~x) + (2) * (y & (~x))
```

Listing 4.7 – Exemple de prédicats opaques injectés

Le *plugin* nous informe également visuellement dans quels blocs de base des prédicats opaques ont été détectés. Par exemple, la Figure 4.6 illustre cette information visuelle. Le bloc de base teint en rouge contient un prédicat opaque dans le calcul de sa condition de branchement. Cela permet à l'analyste de visuellement identifier le code non atteignable et donc de gagner du temps sur la compréhension du programme.

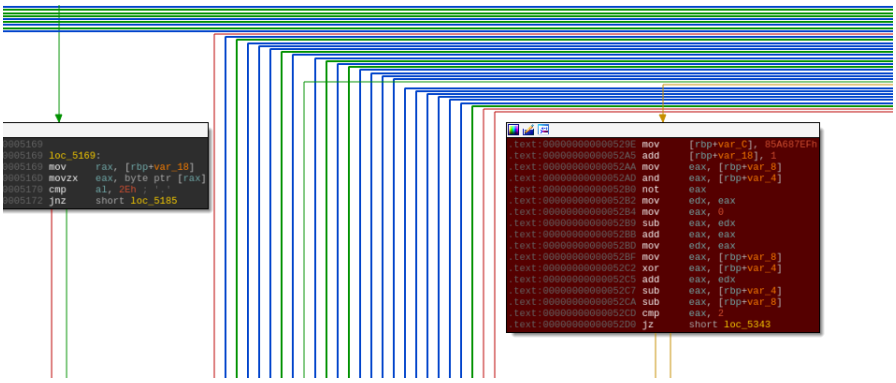

FIGURE 4.6 – Les blocs de base contenant des prédicats opaques sont affichés en rouge pour une reconnaissance visuelle

4.3.2 Deuxième expérimentation

Pour cette deuxième expérimentation nous avons voulu comparer les résultats de notre *plugin* avec ceux d'une étude déjà existante. David et al. [16] ont analysé le *malware* X-Tunnel afin de détecter la présence de prédicats opaques au sein du programme et reconstruisent un CFG simplifié. Pour cette expérimentation nous nous concentrons uniquement sur les résultats obtenus pour la découverte de prédicats opaques et omettons volontairement la reconstruction du CFG.

La version (`99B454`[3]) du *malware* X-Tunnel analysée contient 434143 instructions, 57010 blocs de base et 3488 fonctions pour une taille totale de 1,8 Mo.

	Cond	**PO**	**FP**	**Timeout**	**Temps**
DSE bornée en avant	50 302	7 209	0	472	0h23m

TABLE 4.1 – Résultats d'analyse avec une DSE bornée en avant sur le *malware* X-Tunnel `99B454`

Le Tableau 4.1 illustre les résultats de la détection des prédicats opaques avec notre approche. L'annotation **Cond** représente le nombre d'instructions de branchement testées (ex. adresse d'un `jz`). L'annotation **PO** représente le nombre de prédicats opaques trouvés et **FP** le nombre de faux positifs (vraie condition considérée comme opaque).

Pour identifier les **FP** nous avons suivi la méthode employée par David et al. mais en l'appliquant manuellement et non de façon automatique. Dans leur

3. Version fournie par David et al. afin de travailler sur le même échantillon.

méthode, ils ont déterminé que seules deux formes de prédicat opaque étaient utilisées. Pour chacune des conditions de branchement, ils synthétisent la condition de branchement et effectuent une reconnaissance de schémas connus (*pattern matching*) afin de vérifier si le prédicat opaque possède l'une de ces formes. Dans notre cas, le pattern matching est fait visuellement. Si la condition est taguée comme opaque mais ne correspond pas à l'une des formes pré-identifiée, alors elle est considérée comme un faux positif. Nous confirmons que les deux formes identifiées par David et al. se retrouvent bien dans le binaire. Cependant, nous avons trouvé une nouvelle forme ($x - x = 0$) qui représente 1.05% de la totalité des **PO** trouvés ainsi que 31 autres prédicats opaques (0.43% de la totalité des **PO** trouvés) sans forme particulière ce qui rend les résultats de David et al. un peu approximatifs.

Dans l'étude menée par David et al. les faux négatifs sont calculés en suivant le même principe de synthèse que pour les faux positifs, une condition taguée comme non opaque mais contenant l'une des formes de **PO** pré-identifiée est un faux négatif. Nous considérons que cette synthèse pour la détection des faux négatifs est trop approximative pour être comparée et l'omettons volontairement. Pour pouvoir identifier les faux négatifs, il faudrait connaître toutes les formes de **PO** existantes ou le nombre de **PO** injectés. Information que nous n'avons pas. Le Tableau 4.2 illustre la répartition des formes de **PO** trouvés via les deux approches – sachant que David et al. comptent (à tort) les formes alternatives comme des **FP**.

	$7x^2 - 1 \neq y^2$	$\frac{2}{x^2+1} \neq y^2 + 3$	$x - x = 0$	Unclassified
David et al.	4618 (45.37%)	5560 (54.62%)	n/a	n/a
Notre approche	3197 (44.35%)	3873 (53.72%)	108 (1.05%)	31 (0.43%)

TABLE 4.2 – Répartition des formes de prédicats opaques trouvés dans les deux approches sachant que David et al. comptent (à tort) les formes alternatives comme des FP

La limite de temps consacré à la résolution des contraintes est fixée, dans les deux approches, à 6 secondes. Pour les deux approches le même solveur de contraintes a été utilisé (Z3 [36]) et les expérimentations ont été faites sur des machines Dell classiques.

Les résultats de notre approche sont satisfaisants pour la détection des prédicats opaques : 7209 **PO** avec 0 faux positif. David et al. ont trouvé 9790 **PO** avec 2543 **FP**. Ces résultats sont délicats à comparer compte tenu des défauts de classification **FP**/**FN** de David et al. Nous ne pouvons confirmer que les **PO** trouvés soient les mêmes ou non.

Notons que les deux méthodes sont complémentaires. D'une part notre méthode

prend en compte l'ensemble des instructions de chaque bloc de base et non une sous partie telle que les 16 instructions qui précédent les conditions de branchement (*Backward-Bounded DSE*). D'autre part, la méthode de David et al. permet de détecter les prédicats opaques dont le calcul est dispersé sur plusieurs blocs de base. Par exemple dans X-Tunnel certains prédicats opaques ont une partie de leur calcul dans le prologue d'une fonction X et la deuxième partie dans les différents blocs de base au sein de cette même fonction. Ce type de prédicat opaque n'est pas détectable par notre méthode car nous ne prenons pas en compte l'ensemble du prédicat de chemin. Notre approche peut détecter uniquement les prédicats opaques dont leur calcul est local à un bloc de base. Nous mettons à disposition sur Github [4] l'ensemble des prédicats opaques que nous avons détecté en les classant par groupes de formes.

Notre analyse obtient 472 **Timeout** [5] contre 652 par David et al. Cela peut s'expliquer du fait que nous gardons symbolique uniquement les registres ainsi que les lectures mémoire, toutes les écritures mémoire sont ignorées ce qui facilite la résolution des contraintes. Par ailleurs, noter que nous utilisons la logique SMT QF_BV alors que David et al. utilisent la logique QF_ABV afin de représenter les lectures et écritures mémoire en utilisant un tableau symbolique, ce qui rend le raisonnement plus précis mais plus coûteux.

On peut également constater que notre temps d'analyse (0h23m) est un peu inférieur à celui obtenu par David et al. (0h51m). Cependant, la différence de temps n'est pas représentative car dans leur implémentation, la résolution des contraintes est exportée à travers le réseau sur un serveur distant ce qui augmente bien évidement le temps de traitement.

4.4 Résultats et conclusion

Nous avons montré comment utiliser Triton pour la détection de prédicats opaques dans les conditions de branchement. Cette méthode repose sur un raisonnement symbolique pour vérifier la satisfaisabilité des conditions de branchement à la fin de chaque bloc de base.

Nous avons développé un *plugin* IDA (visible en Annexe et disponible sur Gi-

4. `https://github.com/JonathanSalwan/X-Tunnel-Opaque-Predicates`
5. Après confirmation avec les auteurs, cela comprend également ce qu'ils appellent *unknown*. *Unknown* comprend les *timeout* ainsi que les blocs de base contenant des appels système ne pouvant pas être traités.

thub[6]) permettant de mettre en application la méthode décrite précédemment en combinant Triton avec IDA. IDA a été utilisé pour l'extraction des blocs de base et Triton pour l'exécution symbolique de ces blocs de base. Nous avons également introduit l'utilisation des *hooks* dans Triton afin de contrôler le contexte initial lors de chaque lecture mémoire et ainsi agir sur le comportement des règles symboliques de Triton. Nous avons utilisé ce *plugin* sur des binaires d'expérimentation mais aussi sur la mission initiale (l'analyse d'un programme obscurci propriétaire). L'objectif était de comprendre et d'extraire l'algorithme employé au sein d'une fonction obscurcie. En repérant les prédicats opaques (de l'ordre d'une dizaine), nous avons pu nous concentrer sur les portions de code permettant d'exécuter l'algorithme et d'ignorer celles issues de la protection. Le *plugin* est également toujours utilisé au sein de Quarkslab.

6. `https://github.com/JonathanSalwan/X-Tunnel-Opaque-Predicates`

4.5 Annexes

```
#!/usr/bin/env python2
## -*- coding: utf-8 -*-

import time

from idautils    import *
from idaapi      import *
from idc         import *
from triton      import *

ctx   = None
names = None
nb_bb = 0 # number of basic blocks analyzed
nb_op = 0 # number of opaque predicates found
nb_ed = 0 # number of edges
nb_me = 0 # number of memory access
nb_in = 0 # number of instruction executed
COLOR = 0x000055


def branch(inst, node, startEA):
    global nb_op

    sat   = ctx.isSat(node)
    isOP  = False
    if sat == False and node.isSymbolized():
        isOP = True
        print('[+] Opaque predicat found at 0x%x (always %s)' %(startEA, repr(inst.isConditionTaken())))
        nb_op += 1

    return isOP


def handle_mem_read(ctx, mem):
    global names
    for i in range(mem.getSize()):
        memi = MemoryAccess(mem.getAddress()+i, CPUSIZE.BYTE)
        var = ctx.convertMemoryToSymbolicVariable(memi)
        names.update({var.getName() : str(memi)})


def prove_bb(startEA, endEA):
    global ctx
    global names
    global nb_ed
    global nb_me
    global nb_in

    ctx = TritonContext()
    ctx.setArchitecture(ARCH.X86_64)
    ctx.enableMode(MODE.ALIGNED_MEMORY, True)
    ctx.addCallback(handle_mem_read, CALLBACK.GET_CONCRETE_MEMORY_VALUE)
    names = dict()

    # Symbolize all registers
    for r in ctx.getParentRegisters():
        var = ctx.convertRegisterToSymbolicVariable(r)
        names.update({var.getName() : str(r)})

    ip = startEA
    for _ in range(1000):
        # Fetching opcode from IDA and execute them with Triton
        inst = Instruction()
        opcode = idc.GetManyBytes(ip, 16)
        inst.setOpcode(opcode)
        inst.setAddress(ip)
        ctx.processing(inst)

        # Get next instruction
        ip = ctx.getSymbolicRegisterValue(ctx.registers.rip)

        nb_me += len(inst.getLoadAccess())
        nb_me += len(inst.getStoreAccess())
        nb_in += 1

        # Handle external calls
        if inst.getType() == OPCODE.X86.CALL:
            var = ctx.convertRegisterToSymbolicVariable(ctx.registers.rax)
            names.update({var.getName() : '%s from call at %x' %(str(ctx.registers.rax), ip)})
            ip = inst.getNextAddress()
```

```
        elif inst.isBranch() or ip == endEA:
            nb_ed += 1
            if inst.isBranch() and inst.getType() != OPCODE.X86.JMP:
                nb_ed += 1
            break

    # End of the basic block execution, now get the path predicat
    ast = ctx.getAstContext()
    pc  = ctx.getPathConstraintsAst()

    # Try to get a model for the other branch
    return branch(inst, ast.lnot(pc), startEA)


# Analyze all BB of the current function
func = get_func(ScreenEA())
bbs  = FlowChart(func)
st   = time.time()

for bb in bbs:
    nb_bb += 1
    print('[+] Analyzing basic block at 0x%x' %(bb.startEA))
    isOP = prove_bb(bb.startEA, bb.endEA)
    if isOP:
        for ea in range(bb.startEA, bb.endEA):
            SetColor(ea, CIC_ITEM, COLOR)

et = time.time()

print('[+] Basic Blocks analyzed      : %d' %(nb_bb))
print('[+] Edges analyzed             : %d' %(nb_ed))
print('[+] Instructions analyzed      : %d' %(nb_in))
print('[+] Memory access encountered : %d' %(nb_me))
print('[+] Opaque predicats found     : %d' %(nb_op))
print('[+] Time analysis              : %s seconds' %(et - st))
```

Listing 8 – Plugin IDA + Triton pour la detection de prédicats opaques

Chapitre 5

Dévirtualisation par exécution symbolique

Dans ce chapitre nous présentons une approche permettant la reconstruction de fonctions binaires protégées par virtualisation. Les objectifs de ce chapitre sont :

1. Proposer une nouvelle approche d'analyse pour la reconstruction de fonctions binaires protégées par virtualisation.
2. Donner un deuxième exemple réel d'utilisation de Triton ;
3. Montrer que Triton peut se combiner avec d'autres outils (ex. LLVM [63]) ;

5.1 Introduction

5.1.1 Contexte

La virtualisation est une technique de protection binaire qui consiste à transformer un programme original vers un nouveau jeu d'instructions propriétaire. Ce nouveau jeu d'instructions est ensuite interprété par une machine virtuelle (embarquée dans le binaire) afin de reproduire le comportement d'origine. Cette protection a pour but de cacher le code binaire d'origine ainsi que son CFG, ce qui implique de devoir comprendre, en premier lieu, la machine virtuelle avant de pouvoir reverser le programme ciblé.

Dans le domaine de la dévirtualisation, la plupart des travaux portent sur l'analyse de *malware* afin de reconstruire son flot de contrôle qui peut être vu comme une signature du malware. Dans nos travaux, nous considérons principalement les protections de fonctions sensibles (telles que l'authentification), que ce soit pour des raisons de propriété intellectuelle ou d'intégrité (modification malicieuse). Nous partons du principe que nous avons accès à la fonction virtualisée et nous nous concentrons sur la reconstruction d'une nouvelle version de cette fonction tout en conservant son comportement initial sans tout le processus de la machine virtuelle. Nous considérons les questions ouvertes suivantes :

— Comment peut-on caractériser la pertinence d'un programme dévirtualisé en termes de correction et de précision ?
— À quel point ce genre d'approche est indépendant des différents mécanismes de virtualisation et des protections mises en place pour protéger cette virtualisation contre la rétro-ingénierie ?
— Quelles contre-mesures pourraient permettre de pallier les attaques de dévirtualisation ?

5.1.2 Protection logicielle

La protection logicielle (telle que l'obfuscation [71]) est un enjeu important par exemple dans la lutte pour la protection des propriétés intellectuelles contenues dans les programmes informatiques. En effet, il est possible d'analyser un programme, de comprendre son fonctionnement et donc de pouvoir le reproduire ou même le détourner (ex. passer les tests de sécurité ou de licence). Cela pose problème quand une société investit plusieurs millions d'euros dans sa `R&D` et qu'une société tierce récupère ses algorithmes et donc sa propriété intellectuelle en quelques semaines d'analyse.

Si nous prenons pour exemple une société de jeux vidéo, une très grande partie de ses bénéfices se fait lors de la sortie d'un jeu attendu. Si ce dernier se retrouve accessible sur internet dans les 24 heures qui suivent sa sortie, la société perd une partie de ses bénéfices. La protection logicielle vise justement à rendre compliquée l'analyse d'un programme afin de ralentir au maximum celui qui l'attaque. Dans l'exemple de la sortie d'un jeu vidéo, l'objectif serait de faire en sorte que la protection tienne au moins 20 jours, le temps que la société puisse tirer profit de son produit.

Vous l'aurez compris, une protection logicielle est faite pour ralentir et non pas

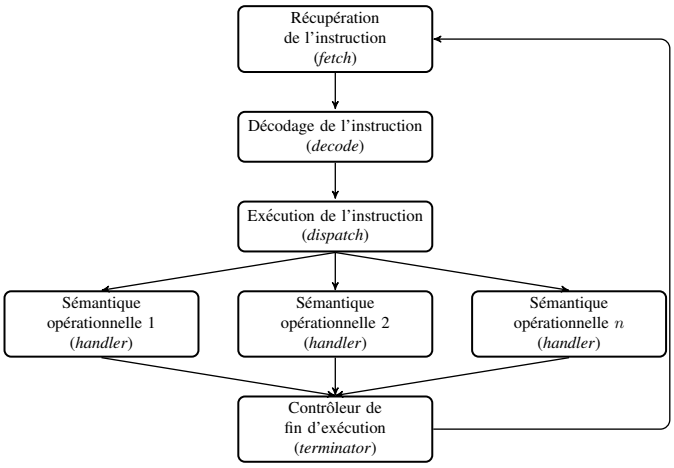

FIGURE 5.1 – Architecture classique d'une machine virtuelle

garantir à 100% la sécurité d'un programme. C'est pourquoi tester une protection se fait en évaluant le temps et les compétences nécessaires à un individu pour casser la protection. Notons également que la dévirtualisation est utile pour l'analyse de malware.

5.1.3 Protection par virtualisation

La virtualisation [45, 9] est une technique de protection binaire qui consiste à transformer le comportement d'un programme original vers un jeu d'instructions (ISA - Instruction Set Architecture) propriétaire puis interprété par une machine virtuelle. Il existe de nombreux outils permettant de virtualiser du code source tels que VMProtect [4], CodeVirtualizer [1], Themida [2] ou encore Tigress [3].

L'architecture la plus commune d'une machine virtuelle (VM) est proche de celle d'un CPU (d'où son nom), voir Figure 5.1. Elle est composée de 4 composants [1] qui constituent l'architecture de l'interpréteur ainsi que d'une série de fonctions (les *handlers*) décrivant la sémantique opérationnelle du jeu d'instructions. La machine virtuelle commence par récupérer (*fetch*) l'instruction courante pointée par son VPC (*Virtual Instruction Pointer*), la décode, exécute sa sémantique opérationnelle (*dispatch* et *handler*), puis détermine si l'exécution doit continuer ou non (*terminator*).

Pour illustrer le rôle que joue une machine virtuelle dans la réorganisation

1. fetch, decode, dispatch, terminator

structurelle d'un code source, prenons comme exemple l'échantillon de code illustré par le Listing 5.1.

Listing 5.1 – Échantillon de code source

```
int f(void) {
  int a = 1;
  int b = 2;
  int c = a * b;
  return c
}
```

Une fois cet échantillon transformé dans le nouveau jeu d'instructions de la machine virtuelle sous la forme de *bytecode*, nous pouvons avoir une séquence binaire qui ressemble à celle illustrée par le Listing 5.2.

Listing 5.2 – Échantillon de bytecode dans le nouveau ISA

```
31 00 01 00 31 01 02 00 44 00 00 01 60
```

Le bytecode est ensuite interprété par la machine virtuelle. Le code illustré par le Listing 5.3 est un exemple simple de machine virtuelle avec un jeu de 4 instructions (`ADD`, `MOV`, `MUL` `RET`) permettant d'interpréter le bytecode du Listing 5.2. La machine virtuelle commence par récupérer la première instruction (`31 00 01 00`), la décode pour déterminer que c'est une instruction de type `MOV`, prépare ses opérandes (`r0, 1`) puis appelle le bon handler (`case MOV`), modifie le VPC (`vpc += 4`) puis revient sur les parties fetch et decode qui récupèrent et décodent la deuxième instruction. L'opération est répétée tant qu'il reste du bytecode à interpréter.

Listing 5.3 – Example de machine virtuelle

```
void vm(ulong vpc, struct vmgpr* gpr) {
  while (terminate() == false) {
    /* Fetch and Decode */
    struct instop* inst = decode(fetch(vpc));
    /* Dispatch */
    switch (inst->getType()) {
      /* Handlers */
      case ADD: /* opcode 0x21 */
        gpr->r[inst->dst] = inst->op[1] + inst->op[2];
        vpc += 4;
        break;
      case MOV: /* opcode 0x31 */
        gpr->r[inst->dst] = inst->op[1];
        vpc += 4;
        break;
      case MUL: /* opcode 0x44 */
        gpr->r[inst->dst] = inst->op[1] * inst->op[2];
        vpc += 4;
        break;
      case RET: /* opcode 0x60 */
        vpc += 1;
```

```
            break;
        }
    }
}
```

L'usage d'une machine virtuelle pour interpréter le comportement d'un programme initial vise à masquer le flot de contrôle de ce dernier et ainsi rendre son analyse plus compliquée.

Ce qui fait la différence entre une bonne et une mauvaise machine virtuelle c'est donc son efficacité à cacher et à rendre complexe l'analyse de ses composants. Par exemple, le bytecode chargée en mémoire peut être chiffré. Les handlers d'instructions peuvent également faire l'usage de *Mixed Boolean-Arithmetics* (MBA [102]) afin de complexifier la compréhension de leur calcul. Il existe également des cas où une machine virtuelle interprète une autre machine virtuelle et ainsi de suite, sur plusieurs niveaux de virtualisation. Par exemple, VMProtect et Tigress proposent d'appliquer plusieurs niveaux de virtualisation sur un code source. Nous reviendrons sur les différents mécanismes de protection dans la Section 5.3.1.

5.1.4 Le Challenge en Reverse Engineering

Le but final pour un analyste peut être multiple (ex : comprendre la structure de la VM, extraire le bytecode, comprendre les handlers...) mais généralement il est de créer un désassembleur du bytecode de la machine virtuelle afin de comprendre le comportement initial du programme. Dans un premier temps l'analyste essaie de trouver où est situé le bytecode du programme virtualisé puis où il est lu pour la première fois par la machine virtuelle. Une fois cette partie trouvée, la partie fetch de la machine virtuelle est identifiée. En suivant le flot de contrôle de la machine virtuelle, nous arrivons à la partie decoding. Il est nécessaire de comprendre dans son intégralité comment une instruction est décodée. Par exemple, il faut pouvoir être en mesure de lister tous les types d'instruction et opérandes que la VM peut utiliser. Une fois le decoding compris, et en suivant le flot de contrôle, nous arrivons sur la partie qui "dispatch" le décodage d'une instruction sur le handler approprié. Il est important de comprendre comment ce dispatcheur orchestre les liens entre le décodage d'une instruction et son handler. Une fois ce lien établi, il ne reste plus qu'à comprendre le calcul effectué dans les handlers pour donner un nom syntaxique aux instructions décodées et ainsi pouvoir développer un désassembleur. Par exemple, comprendre que le décodage de `44 00 00 01` appelle l'handler `MUL` et que cet handler effectue une multiplication sur le registre r0 et r1 en plaçant le résultat dans r0. Ainsi, on sait que le bytecode `44 00 00 01` peut se désassembler

en `mul r0, r0, r1`.

Pour résumer, les challenges sont :

1. Identifier que le binaire est virtualisé et identifier ses entrées ;
2. Identifier chacun des composants de la machine virtuelle ;
3. Comprendre comment tous ces composants sont liés les uns aux autres. Plus particulièrement : quel handler est appelé pour un bytecode donné, quel calcul appliqué au sein de chaque handler, comment sont encodés les opérandes d'une instruction ;
4. Comprendre comment le VPC est orchestré afin de déterminer comment les instructions s'enchainent ;
5. Une fois que tous les points précédents sont clairs, il reste à développer un désassembleur pour l'ISA de la machine virtuelle pour commencer à analyser le code protégé.

Résoudre chacun des ces points est plus ou moins long en fonction des connaissances de l'analyste, du niveau et de la quantité de protections intégrées pour protéger la machine virtuelle, et de l'architecture de la machine virtuelle.

Approches existantes : Les approches visant à défaire les protections binaires telles que la virtualisation ont émergé durant la dernière décennie. On peut trouver des approches semi-manuelle [68, 80, 84], automatique [60, 80, 88, 98, 99] et de synthèse [57]. Les approches dites semi-manuelle consistent à analyser manuellement les différents composants qui forment une machine virtuelle afin de développer un désassembleur pouvant ensuite être intégré dans des outils tels que IDA [54], Radare [74] ou encore Binary Ninja [7]. L'efficacité de cette approche est fortement liée à l'expertise de l'analyste et elle reste, quoi qu'il arrive, une analyse fastidieuse même pour un expert (approche manuelle propre à chaque VM, non réutilisable de manière automatique sur une autre classe de VM et donc rédeveloppement des outils pour chaque cas d'analyse). Certaines classes d'analyse automatique [60, 88] consistent à implémenter des mécanismes permettant de détecter automatiquement les différents composants d'une machine virtuelle en se basant sur des schémas connus (*pattern matching*). Cependant, ces analyses sont fortement liées à une forme particulière de machine virtuelle et sont rarement applicables quand de nouvelles VMs sont rencontrées. Enfin, il existe une certaine classe d'analyse automatique [32, 98] qui vise à reconstruire directement le comportement du programme d'origine en se basant sur des traces d'exécution et en enlevant le processus de virtualisation.

Discussion : Retrouver 100% du code d'origine d'une fonction virtualisée est une tâche très complexe, pour ne pas dire impossible. C'est pourquoi la dévirtualisation tente de proposer au mieux un code pouvant correspondre à celui d'origine. La question que l'on peut se poser est : *finalement, est-il vraiment nécessaire de retrouver 100% du code d'origine tant que le comportement du programme dévirtualisé est conservé ?*

5.1.5 Contribution

Notre approche est dynamique, automatique et reconstruit, sous la forme binaire, le comportement initial d'un code virtualisé. Pour cela, nous combinons l'analyse de teinte, l'exécution symbolique, la simplification d'expressions symboliques (guidée par l'état de teinte) ainsi que la simplification de code.

	Manuelle	Kinder[60]	Coogan[32]	Yadegari[99]	Notre approche
Comprendre vpc	requis	requis	non	non	non
Comprendre dispatcher	requis	non	non	non	non
Comprendre bytecode	requis	non	non	non	non
Résultat fourni	CFG simplifié	CFG + invariants	trace simplifiée	CFG simplifié	code simplifié
Méthode	_	analyse statique interprétation abstraite	*value-based slicing*	teinte, symbolique, simplification d'instructions	teinte, symbolique, simplification de formules, simplification de code
XP : type de code	_	*toy*	*toys+malware*	*toys+malware*	fonctions pures
XP : #échantillons	_	1	12	44	920
XP : métriques d'eval.	_	invariants connus	%simplification	similarité	taille, correction

FIGURE 5.2 – Positionnement de notre approche

La Figure 5.2 positionne notre approche comparée à celles déjà publiées (nous discutons en détail les différentes approches en Section 5.8). Yadegari et al. [99] poursuivent une approche similaire en combinant analyse de teinte et exécution symbolique, mais nous visons comme résultat la construction d'un code binaire et nous établissons un banc d'expérimentation plus poussé. Chacune des approches mentionnées fournit un résultat d'analyse différent tel que des traces ou des CFG simplifiés. Le point qui diverge également avec les autres approches concerne les caractéristiques intrinsèques des programmes analysés. Notre analyse est axée sur l'analyse de fonction *pures*. On considère une fonction pure si cette dernière ne possède aucun effet de bord. De plus, le résultat de notre analyse est plus précis (voir Section 5.4.1) et plus efficace (voir Section 5.4.2) si la fonction ne possède aucune boucle à borne dynamique, aucun accès mémoire symbolique et peu de chemins. Par exemple, les fonctions de hachage rentrent parfaitement dans cette catégorie de fonctions et correspondent à notre cible d'analyse. C'est pourquoi nous axons nos expérimentations sur les fonctions de hachage virtualisées. Les

limitations de notre approche sont discutées en Section 5.7, en particulier si la fonction analysée ne possède pas les propriétés attendues.

Pour résumer, nos contributions sont :

— Une approche [2] dynamique et automatique permettant la dévirtualisation de code binaire en combinant une analyse de teinte, une exécution symbolique dynamique et une politique de simplification de code. Nous discutons également des limitations et des garanties de notre approche puis nous démontrons le potentiel de cette dernière en résolvant automatiquement (la partie *non-jitted*) le challenge Tigress non résolu publiquement auparavant [3].

— Un banc d'expérimentation [4] permettant d'évaluer notre approche sur plusieurs critères bien définis, pour une catégorie de programmes bien particulières (fonctions pures) et sur plusieurs formes et combinaisons de protection. Ce banc d'expérimentation permet de (1) de reproduire nos résultats, (2) d'utiliser notre approche sur de nouveaux échantillons ainsi que de nouvelles formes de protection.

5.2 Dévirtualisation, notre approche

Comme nous pouvons l'imaginer, le but d'une déobfuscation de protection telle que la virtualisation est de retrouver (au plus proche) le code d'origine d'une fonction protégée (on note f la fonction protégée). Notre approche vise à créer une nouvelle version binaire non virtualisée (que l'on note f') en partant d'une fonction protégée tout en gardant les mêmes propriétés initiales (sans effet de bord) $\forall x,\ f(x) = f'(x)$.

5.2.1 Aperçu de l'analyse

Notre approche repose sur une intuition clef qui est propre au fonctionnement des machines virtuelles. *Une trace obfusquée T' (d'un programme virtualisé P') combine les instructions d'origine du programme P (la trace T correspond à la trace T' dans le programme P) et les instructions de la machine virtuelle VM tel*

2. Publiée à DIMVA 2018 [83].
3. `http://tigress.cs.arizona.edu/solution-0004.html`
4. La première version de ces expérimentation ont été présenté au SSTIC 2017 [5].

que $T' = T + VM(T)$. Si nous arrivons à distinguer les deux séquences d'instructions T et $VM(T)$, nous sommes en mesure de reconstruire un chemin du programme d'origine P depuis une trace T'. En répétant cette opération sur tous les chemins du programme virtualisé, nous serons en mesure de reconstruire le programme d'origine P – dans le cas où le programme d'origine possède un nombre fini de chemins exécutables et que nous pouvons les énumérer, ce qui correspond généralement à la classe de fonctions auxquelles on s'intéresse.

Nous décrivons ici les différentes étapes de notre approche.

Étape 1 - Analyse par teinte : La première étape vise à séparer les instructions qui font partie de la machine virtuelle et celles qui sont exécutées pour simuler le fonctionnement du programme d'origine, nous appelons ces dernières *instructions pertinentes*. Pour cela nous teintons les entrées de la fonction virtualisée et nous exécutons cette fonction avec des valeurs aléatoire pour ces entrées. À la fin de cette première étape nous obtenons une sous-trace d'instructions teintées (représentant l'ensemble des instructions pertinentes) du chemin emprunté. À ce stade, le comportement du programme d'origine (pour une entrée donnée) est défini par cette sous-trace d'instructions teintées. Cependant, cette sous-trace ne peut pas être rejouée (au sens réexécutée) car certaines valeurs sont absentes telles que les valeurs initiales des registres (car non teintées).

Étape 2 - Exécution symbolique : La deuxième étape est appliquée en parallèle de l'étape 1 et consiste à (1) représenter l'exécution du programme virtualisé, pour une entrée donnée, par un prédicat de chemin dans lequel les entrées teintées lors de l'étape 1 sont symbolisées, (2) appliquer l'optimisation `ONLY_ON_TAINTED` afin de concrétiser toutes les expressions non teintées présentes dans le prédicat de chemin et ainsi omettre le calcul de la machine virtuelle, (3) effectuer une couverture de code lors de l'étape 3.

Étape 3 - Construction de l'arbre de d'exécution : Afin de retrouver le comportement complet d'une fonction virtualisée, nous devons prendre en compte l'ensemble de ses chemins. Pour cela, nous effectuons une couverture de chemins basée sur le prédicat de chemin construit lors de l'étape 2 et nous construisons un arbre d'exécution représentant l'ensemble des chemins de la fonction virtualisée.

Étape 4 - Construction du binaire de la fonction : Pour finir, nous convertissons l'arbre d'exécution de la fonction virtualisée vers celui d'une plateforme de compilation afin de pouvoir construire une version non virtualisée de la fonction protégée et ainsi faciliter la compréhension du code initialement virtualisé.

L'ensemble de ces étapes, ainsi que leur chronologie, sont illustrées par la Figure 5.3 et sont détaillées dans les sections suivantes. Noter que dans notre approche, l'étape 0 (identification des entrées de la fonction qui seront teintées) doit toujours être effectuée manuellement et de manière traditionnelle. Sont considérées comme entrées toutes interactions externes avec l'utilisateur, telles que les variables d'environnement, les arguments du programme et les appels système (exemple : *read*, *recv*...). Les analystes s'appuient généralement sur des outils tels que IDA ou GDB pour identifier ces entrées.

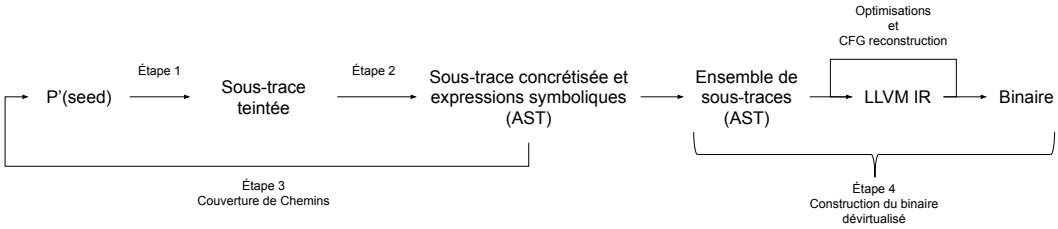

FIGURE 5.3 – Schéma global de l'approche de dévirtualisation

5.2.2 Prérequis

Notre approche ne nécessite pas la connaissance des composants de la machine virtuelle. Cependant elle nécessite 3 prérequis :

— L'identification qu'une partie du programme est virtualisée ;
— La localisation dans le programme de la fonction virtualisée, afin de définir des limites d'analyse et de reconstruction ;
— Les entrées de la fonction virtualisée afin d'initialiser la teinte.

Nous allons désormais détailler les différentes étapes de l'approche.

5.2.3 Étape 1 - Analyse par teinte

Cette étape consiste à isoler les instructions qui font partie de la machine virtuelle et celles qui sont exécutées pour simuler le comportement d'origine de la fonction virtualisée. Afin de bien comprendre cette étape prenons comme exemple la fonction f du Listing 5.4 qui effectue un `xor` sur son argument.

```
int f(int x) {
    return x ^ 0x20;
}
```

Listing 5.4 – Échantillon de code source

Après virtualisation nous obtenons une fonction (f') qui embarque une machine virtuelle et qui simule, en fonction de son entrée, le comportement de la fonction f. Le CFG de cette fonction virtualisée est illustré par la Figure 5.4.

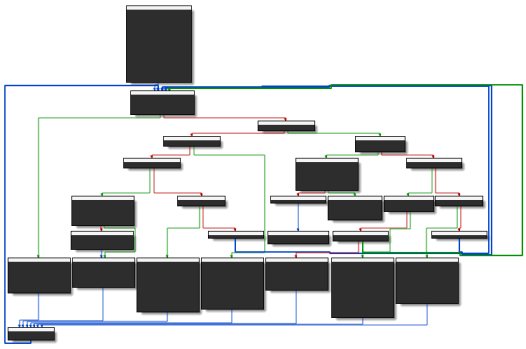

FIGURE 5.4 – Virtualisation du Listing 5.4
Virtualisation avec Tigress

Si nous exécutons cette version protégée de la fonction f' avec une entrée aléatoire (disons 100) et que nous comptons le nombre d'instructions exécutées sur la trace T', nous obtenons 263 instructions contre seulement 7 pour la trace d'origine T. Maintenant si nous teintons l'argument de la fonction f' et que nous isolons les instructions teintées sur la trace T' nous obtenons 11 instructions (voir Listing 5.5). Ces instructions sont ce que nous appelons les instructions pertinentes issue de la trace virtualisée T'. Cette sous-trace représente donc les instructions de la machine virtuelle permettant de rejouer le comportement du programme initial. Par exemple, nous pouvons y voir qu'une instruction `XOR` est bien exécutée.

Cependant, nous ne pouvons pas extraire cette sous-trace et l'exécuter de façon autonome en espérant rejouer le comportement du Listing 5.4. Premièrement, cette

sous-trace ne représente pas l'exécution d'un ensemble d'instructions contigües (les instructions teintées se trouvent à différents endroits dans la trace T'). Deuxièmement, pour chacune des instructions teintées, il nous faut connaître l'état initial des registres et de la mémoire. Par exemple, nous ne voyons aucune référence à la constante `0x20` présente dans le Listing 5.4.

```
0x6e2: mov dword ptr [rbp - 0x124], edi
...
0x82c: mov edx, dword ptr [rdx]
0x82e: mov dword ptr [rax], edx
...
0x85d: mov ecx, dword ptr [rax]
...
0x868: xor eax, ecx
0x86a: mov dword ptr [rdx], eax
...
0x8aa: mov edx, dword ptr [rdx]
0x8ac: mov dword ptr [rax], edx
...
0x82c: mov edx, dword ptr [rdx]
0x82e: mov dword ptr [rax], edx
...
0x984: mov eax, dword ptr [rax]
```

Listing 5.5 – Sous-trace illustrant les instructions pertinentes de la trace T'

5.2.4 Étape 2 - Exécution symbolique

Cette étape consiste à représenter l'exécution du programme virtualisé, pour une entrée donnée, en un prédicat de chemin dans lequel les entrées teintées lors de l'étape 1 sont symbolisées. En appliquant l'optimisation `ONLY_ON_TAINTED` (voir Section 3.6.2), Triton concrétise toutes les expressions non teintées présentes dans le prédicat de chemin. Ce qui revient naturellement à omettre le comportement de la machine virtuelle et de garder uniquement le comportement du programme d'origine.

Pour bien comprendre l'étape de concrétisation qui est l'étape la plus importante dans le processus de dévirtualisation, nous rappelons le fonctionnement de l'optimisation `ONLY_ON_TAINTED` (voir Section 3.6.2). Prenons comme exemple le code illustré par le Listing 5.6.

```
int f(int x) {
    int var1 = 1;
    int var2 = 2;
    int var3 = var1 + var2;
```

```
    int var4 = 6;
    int var5 = 3;
    int var6 = var4 ^ var5;
    int var7 = x + var6;
    int var8 = var3 * var7;
    return var8;
}
```

Listing 5.6 – Échantillon de code source

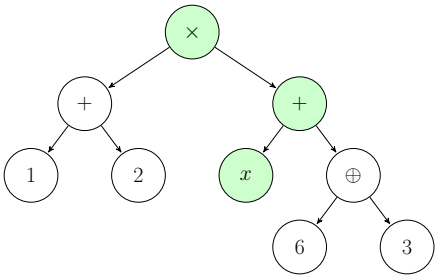

FIGURE 5.5 – Arbre affecté à la variable `var8` au retour de la fonction f sans l'optimisation `ONLY_ON_TAINTED`

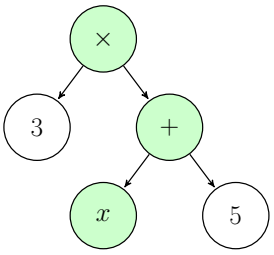

FIGURE 5.6 – Arbre affecté à la variable `var8` au retour de la fonction f avec l'optimisation `ONLY_ON_TAINTED`

La variable x de la fonction f est teintée et symbolisée. Puis nous exécutons la fonction avec pour entrée une valeur aléatoire. Une fois la fonction exécutée, l'arbre représentant l'expression symbolique affecté à la variable `var8` au retour de la fonction est illustré par la Figure 5.5. Les nœuds verts sont les nœuds issus de la teinte de x.

Lors de l'exécution de la fonction f, l'optimisation `ONLY_ON_TAINTED` concrétise les expressions non teintées (nœuds blancs). Avec cette optimisation activée, l'arbre affecté à la variable `var8` au retour de la fonction est illustré par la Figure 5.6. Les deux arbres représentent le même calcul mais non pas la même taille.

Si cette fonction f est virtualisée, nous aurions un arbre beaucoup plus gros où les nœuds blancs représenteraient le calcul de la machine virtuelle. Ainsi, en appliquant l'`ONLY_ON_TAINTED` cela nous permet donc d'omettre le calcul de la machine virtuelle et de conserver uniquement le calcul du programme d'origine, ce qui résulte en une dévirtualisation.

5.2.5 Étape 3 - Construction de l'arbre d'exécution

Afin d'extraire le comportement complet d'une fonction virtualisée, nous devons représenter l'ensemble de ses chemins. Pour cela nous construisons un arbre d'exécution représentant l'ensemble des chemins couverts de la fonction.

Représentation d'une fonction : Nous représentons une fonction sous la forme d'un arbre qui lie chaque valeur d'entrée à une valeur de sortie. Chaque nœud racine est un opérateur et les feuilles les opérandes. Le tout forme une expression représentant le calcul réalisé par la fonction. Nous appelons cet arbre *l'arbre d'exécution d'une fonction.*

```
int f(int x) {
    if (x == 10)
      return 1;
    else
      return 0;
}
```

Listing 5.7 – Exemple de fonction avec une condition de branchement

```
int f(int x) {
    if (x > 0) {
      if (x == 10)
          return 1;
      return 2;
    }
    else
      return 0;
}
```

Listing 5.8 – Exemple de fonction avec plusieurs conditions de branchement

Les Figures 5.7 et 5.8 représentent respectivement les arbres d'exécution des Listings 5.7 et 5.8. Pour une meilleure compréhension, nous avons mis les états de sortie d'une fonction en bleu et ses contraintes en vert. Par exemple, l'arbre de la Figure 5.7 se lit comme suit : *La fonction* $f(x)$ *vaut* 1 *si* x *est égale à* 10 *sinon elle vaut* 0.

5.2.5.1 Étape 3a - Retrouver les chemins

Dans cette étape nous appliquons une couverture de chemins comme décrite dans la Section 3.2.3.4. À la fin de chaque exécution permettant de couvrir un nouveau chemin, nous obtenons un prédicat de chemin. Dans le cas du Listing 5.8, une fois la fonction `f` couverte complètement, nous obtenons donc trois prédicats de chemin que nous allons fusionner.

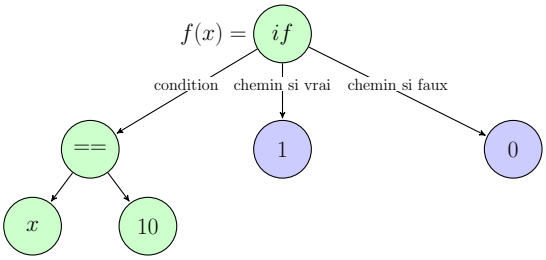

FIGURE 5.7 – Arbre d'exécution de la fonction illustrée par le Listing 5.7

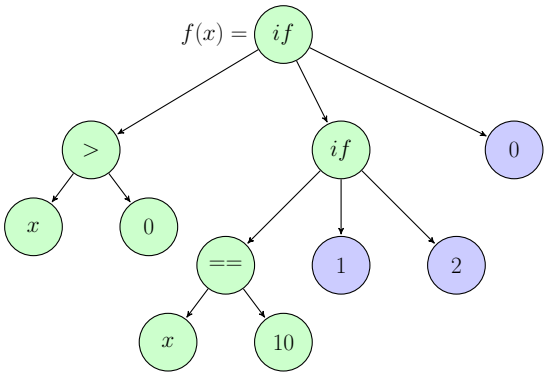

FIGURE 5.8 – Arbre d'exécution de la fonction illustrée par le Listing 5.8

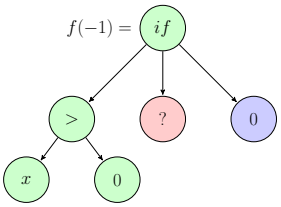

FIGURE 5.9 – Arbre d'exécution partiel pour $f(-1)$

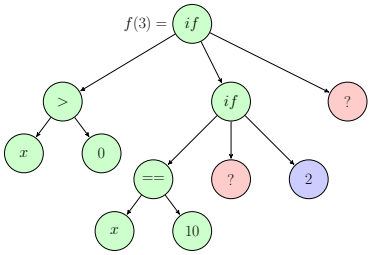

FIGURE 5.10 – Arbre d'exécution partiel pour $f(3)$

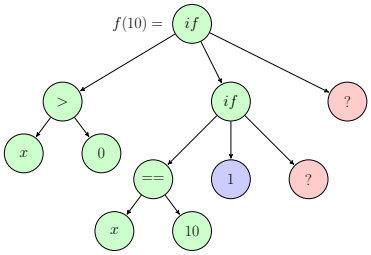

FIGURE 5.11 – Arbre d'exécution partiel pour $f(10)$

5.2.5.2 Étape 3b - Fusion des arbres d'exécution

Cette étape consiste à fusionner l'ensemble des arbres d'exécution partiels récupérés en étape 3a afin d'obtenir un arbre d'exécution complet représentant l'ensemble des chemins d'une fonction. Pour cela nous introduisons la notion de nœud non couvert dans nos arbres (les nœuds rouges dans les Figures 5.9, 5.10 et 5.11). Notre algorithme de fusion est simple : on note L notre liste d'arbres d'exécution partiels et nous prenons comme base de construction un arbre aléatoire A dans cette liste. Nous remplaçons les nœuds rouges présents dans A par les nœuds issus des arbres de la liste L permettant de couvrir cette branche. L'opération est répétée tant qu'il reste des nœuds rouges dans A.

5.2.6 Étape 4 - Construction du binaire de la fonction

Afin de pouvoir fournir une version binaire de la fonction analysée, il est nécessaire de convertir notre représentation sous forme d'arbre, vers une architecture spécifique (e.g : `x86`). Pour cela, nous convertissons notre représentation vers celle d'une des infrastructures de compilation. Nous avons fait le choix de LLVM [63] pour plusieurs raisons :

1. La représentation des arbres de Triton est un sous-ensemble de l'IL-LLVM ce qui nous permet d'appliquer une conversion facilement (*one-to-one*) ;
2. Nous pouvons bénéficier du *front-end* de LLVM afin de compiler notre représentation vers différentes architectures telles que `x86`, `ARM`, `Sparc`, `MIPS`, `PowerPC`, etc ;
3. Une fois notre représentation convertie vers celle de LLVM, nous pouvons utiliser les simplifications de LLVM afin de " polir " l'assembleur généré (factorisation des chemins, optimisation du code, etc.).

Note : On discutera des limitation de notre approche en Section 5.7.

5.2.7 Implémentation de l'approche

Nous utilisons un script [5] basé sur LIEF [91] pour l'extraction d'information (segments exécutables, symboles...) dans des formats binaires tels que PE, ELF, Mach-O (point 1 Figure 5.12). Une fois que les informations concernant le binaire protégé sont récupérées, elles sont envoyées à Triton [5] afin d'exécuter le code et y appliquer les analyses de teinte et l'exécution symbolique (point 2 Figure 5.12). Pour la conversion de représentation entre l'IR de Triton et l'IR d'LLVM [63] nous avons utilisé Arybo [42] (points 3 et 4 Figure 5.12) et enfin le *front-end* d'LLVM pour compiler la nouvelle version non virtualisée du binaire protégé (point 5 Figure 5.12).

5. `https://github.com/JonathanSalwan/Tigress_protection/blob/master/solve-vm.py`

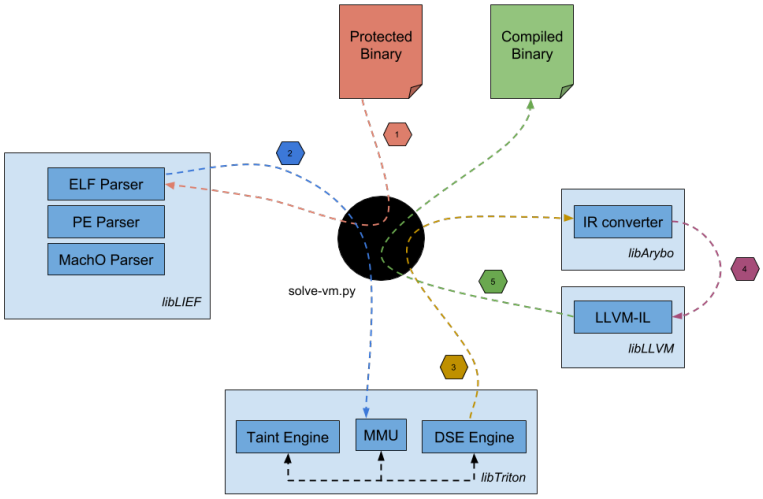

FIGURE 5.12 – Chronologie d'exécution entre les composants de notre implémentation

5.3 Environnement d'expérimentations

Afin d'évaluer notre approche nous avons mené deux expérimentations. La première dans un environnement où nous contrôlons les sources à protéger et les options de protection. La deuxième dans un environnement non contrôlé par le biais de challenges publiques (le challenge Tigress [6]). Nos travaux ont été présentés au SSTIC [82] et à DIMVA [83].

Le résultat des métriques ainsi que les échantillons et les scripts pouvant rejouer les tests sont disponibles sur github [7]. Afin d'évaluer notre approche nous définissons trois critères :

- C_1 : **Précision**, à quel point notre approche est-elle exacte ?
- C_2 : **Efficacité**, à quel point notre approche est-elle efficace et passe-t-elle à l'échelle ?
- C_3 : **Robustesse**, à quel point les protections influencent-elles notre approche ?

6. `http://tigress.cs.arizona.edu/challenges.html`
7. `https://github.com/JonathanSalwan/Tigress_protection`

5.3.1 Installation de l'environnement contrôlé

Notre banc de test est composé de 20 fonctions de hachage. Sur ces 20 fonctions, 10 d'entre elles sont publiquement connues [8] et 10 autres sont issues du challenge Tigress [9]. Le Tableau 5.1 liste les caractéristiques de chacune de ces fonctions en termes de taille et de nombre de chemins dépendant de l'entrée utilisateur. Les fonctions proposées sont typiquement le genre de fonction qu'un développeur pourrait vouloir protéger dans un processus d'identification ou de vérification d'intégrité.

Hash	Loops	Binary Size (inst)	Paths input dependent
Adler-32	✓	78	1
CityHash	✓	175	1
Collberg-0001-0	✓	167	1
Collberg-0001-1	×	177	2
Collberg-0001-2	×	223	1
Collberg-0001-3	✓	195	1
Collberg-0001-4	✓	183	1
Collberg-0004-0	×	210	2
Collberg-0004-1	×	143	1
Collberg-0004-2	✓	219	2
Collberg-0004-3	✓	171	1
Collberg-0004-4	✓	274	1
FNV1a	×	110	1
Jenkins	✓	79	1
JodyHash	✓	90	1
MD5	✓	314	1
SpiHash	✓	362	1
SpookyHash	✓	426	1
SuperFastHash	✓	144	1
Xxhash	✓	182	1

Table 5.1 – Caractéristiques des fonctions de hachage pour notre banc de test

Afin de protéger nos fonctions de hachage, nous avons choisi un outil de virtualisation niveau source en libre utilisation nommé Tigress [10]. Tigress est une machine virtuelle / obfuscateur pour le langage C qui propose des protections contre les attaques de *reverse engineering* statiques et dynamiques.

8. `https://en.wikipedia.org/wiki/List_of_hash_functions`
9. Merci à Christian Collberg de nous avoir fourni les sources.
10. `http://tigress.cs.arizona.edu`

Afin de pouvoir intégrer et analyser de façon automatique tous nos échantillons, nous les avons structurés d'une façon identique. Ils possèdent tous une fonction intitulée `secret` qui prend un entier en paramètre et qui renvoie le dérivé de cet entier en fonction de l'algorithme de hachage utilisé (voir Listing 5.9).

```
long secret(long input) {
  /* Algorithme de hachage sur input */
  return input;
}
```

Listing 5.9 – Template pour nos échantillons

Pour chacune des fonctions `secret`, nous appliquons 46 protections de virtualisation différentes. Chacune de ces protections produit un nouveau binaire protégé. Ce qui, au final, nous donne pour notre banc de test un total de 920 échantillons protégés (20 x 46).

5.3.2 Détail des protections utilisées

Le détail des protections utilisées durant nos expérimentations est présenté ci-dessous :

- **Anti Branch Analysis** (options : goto2push, goto2call, branchFuns) : Le rôle de cette transformation est de rendre difficile l'analyse de la destination des instructions de branchement. Par exemple, sur une architecture `x86` au lieu d'effectuer un `call 0x11223344`, il est possible d'effectuer un `push eax ; ... ; ret` ou `eax` contient l'adresse `0x11223344` issue d'une opération arithmétique.

- **Max Merge Length** (options : 0, 10, 20, 30) : Le rôle de cette transformation est d'unir le comportement de plusieurs instructions virtualisées dans un même handler d'instruction afin de rendre compliqué la compréhension de ces dernières.

- **Bogus Function** (options : 0, 1, 2, 3) : Le rôle de cette transformation est de rendre compliqué la détection du VPC. Par exemple le calcul du VPC est dispersé à travers le programme, certains de ces calculs sont fictifs et d'autres réels.

- **Kind of Operands** (options : stack, registers) : Le rôle de cette transformation est fournir deux façons de gérer les opérandes pour les instructions virtualisées. Par exemple, il est possible de passer les opérandes depuis la pile ou de les passer via les registres. Il est également possible de mixer ces deux méthodes afin d'avoir une gestion des opérandes aléatoire.

- **Opaque to VPC** (options : true, false) : Cette transformation utilise des prédicats opaques sur le calcul du VPC afin de rendre compliqué la compréhension de celui-ci.

- **Bogus Loop Iterations** (options : 0, 1, 2, 3) : Le rôle de cette transformation est de rajouter des calculs aléatoires et inutiles lors de chaque itération du dispatcheur. Cela a pour objectifs de : (a) brouiller la compréhension du dispatcher ; (b) augmenter la longueur des traces afin de les rendre compliquées à enregistrer et analyser.

- **Super Operator Ratio** (options : 0, 0.2, 0.4, 0.6, 0.8, 1.0) : Le rôle de cette transformation est de rendre la sémantique des instructions compliquée à comprendre en rajoutant plusieurs opérations arithmétiques et en combinant les opérandes des instructions avec la pile et les registres. Cela a pour but de rendre compliquée l'analyse des interpréteurs comme discuté par R.Rolles [80].

- **Random Opcodes** (options : true, false) : Le rôle de cette transformation est de rendre ou non aléatoire l'attribution des opcodes de la machine virtuelle à la compilation. Cela rend un désassembleur créé par l'attaquant spécifique au binaire compilé. Par exemple, si nous avions découvert, sur un binaire protégé A, que le bytecode `00 11 22 33` effectuait un `add r1, r2`, cela ne serait pas vrai pour un binaire protégé B.

- **Duplicate Opcodes** (options : 0, 1, 2, 3) : Le rôle de cette transformation est de dupliquer les opcodes tout en gardant la même sémantique. Par exemple, une instruction `ADD` pourrait utiliser ses opérandes via la pile tandis qu'une autre instruction `ADD` les utiliserait via les registres. Lorsqu'une autre pourrait combiner les deux. Tigress choisit de façon aléatoire les combinaisons possibles.

- **Dispatcher** (options : binary, direct, call, interpolation, indirect, switch,

ifnest, linear) : Le rôle de cette transformation est d'avoir des dispatcheurs d'exécution différents afin de rendre compliqué la détection du VPC.

- **Encode Byte Array and Obfuscate Decoder** (options : true, false) : Le rôle de cette transformation est de ne pas avoir le bytecode du code virtualisé en clair dans le programme. Par exemple, le bytecode est chiffré à la compilation et déchiffré lors de l'exécution.

- **Nested VMs** (options : 1, 2, 3) : Le rôle de cette transformation est de définir le nombre de niveaux de virtualisation imbriqués. Il est possible d'avoir une machine virtuelle dans une autre et ainsi de suite.

5.3.3 Matériel utilisé pour les expérimentations

Toutes les expérimentations on été faites sur un ordinateur portable Dell XPS 13 avec un CPU Intel i7-6560U, 16Go de RAM avec 8Go de SWAP SSD. Noter également que tous les résultats et scripts utilisés pour les métriques sont disponibles en libre accès [11].

5.4 Première expérimentation

Nous présentons dans cette section la première expérimentation sur l'environnement contrôlé et discutons les résultats des critères C_1, C_2 et C_3.

5.4.1 Précision (C_1)

Objectif : Le critère C_1 vise à répondre à deux points spécifiques (a) la **correction,** est-ce que le binaire dévirtualisé possède toujours les mêmes propriétés sémantiques que le binaire obfusqué ? (b) la **concision,** est-ce que le binaire dévirtualisé possède un nombre d'instructions proche de celui d'origine ?

Métriques utilisées : Pour ce qui concerne la correction, une fois l'approche appliquée et le binaire dévirtualisé, nous testons les deux binaires avec pour entrée

11. `https://github.com/JonathanSalwan/Tigress_protection`

	Traces Size (instructions)			Binary Size (instructions)					
Hash	**Original**	**Obfuscated**	**Deobfuscated**	**Original**	**Obfuscated**	**Deobfuscated**	**Time (s)**	**RAM (KB)**	**Correctness**
Adler-32	min : 235 max : 235 avg : 235	min : 5,385 max : 2,996,678 avg : 169,174	min : 222 max : 222 avg : 222	min : 78 max : 78 avg : 78	min : 665 max : 2,001 avg : 1,092	min : 222 max : 222 avg : 222	min : 0.4 max : 516.0 avg : 26.6	min : 84,784 max : 2,469,276 avg : 203,737	100%
CityHash	min : 200 max : 200 avg : 200	min : 1,455 max : 37,532 avg : 3,555	min : 57 max : 57 avg : 57	min : 175 max : 175 avg : 175	min : 571 max : 1,396 avg : 938	min : 57 max : 57 avg : 57	min : 0.1 max : 3.2 avg : 0.3	min : 81,664 max : 93,540 avg : 82,756	100%
Collberg-0001-0	min : 173 max : 173 avg : 173	min : 4,497 max : 2,840,513 avg : 174,703	min : 79 max : 79 avg : 79	min : 167 max : 167 avg : 167	min : 679 max : 3,366 avg : 1,243	min : 79 max : 79 avg : 79	min : 0.4 max : 494.7 avg : 26.4	min : 86,008 max : 2,339,380 avg : 204,447	100%
Collberg-0001-1	min : 326 max : 326 avg : 326	min : 8,456 max : 2,066,306 avg : 103,599	min : 167 max : 167 avg : 167	min : 177 max : 177 avg : 177	min : 685 max : 3,697 avg : 1,300	min : 96 max : 96 avg : 96	min : 0.8 max : 184.2 avg : 9.3	min : 102,948 max : 883,780 avg : 136,964	100%
Collberg-0001-2	min : 227 max : 227 avg : 227	min : 7,099 max : 2,132,169 avg : 104,401	min : 84 max : 84 avg : 84	min : 223 max : 223 avg : 223	min : 685 max : 5,043 avg : 1,364	min : 84 max : 84 avg : 84	min : 0.6 max : 182.0 avg : 9.3	min : 93,016 max : 899,528 avg : 127,183	100%
Collberg-0001-3	min : 262 max : 262 avg : 262	min : 7,467 max : 2,071,933 avg : 98,637	min : 68 max : 68 avg : 68	min : 195 max : 195 avg : 195	min : 687 max : 4,367 avg : 1,342	min : 68 max : 68 avg : 68	min : 0.6 max : 164.8 avg : 8.0	min : 90,400 max : 898,128 avg : 122,732	100%
Collberg-0001-4	min : 228 max : 228 avg : 228	min : 5,881 max : 1,510,030 avg : 107,831	min : 100 max : 100 avg : 100	min : 183 max : 183 avg : 183	min : 709 max : 3,676 avg : 1,315	min : 100 max : 100 avg : 100	min : 0.5 max : 168.1 avg : 12.5	min : 86,564 max : 896,148 avg : 145,051	100%
Collberg-0004-0	min : 372 max : 372 avg : 372	min : 10,348 max : 7,804,232 avg : 465,758	min : 190 max : 190 avg : 190	min : 210 max : 210 avg : 210	min : 702 max : 3,631 avg : 1,317	min : 99 max : 99 avg : 99	min : 1.1 max : 1431.0 avg : 74.4	min : 116,452 max : 6,306,932 avg : 435,567	100%
Collberg-0004-1	min : 147 max : 147 avg : 147	min : 3,810 max : 859,278 avg : 46,031	min : 67 max : 67 avg : 67	min : 143 max : 143 avg : 143	min : 636 max : 2,704 avg : 1,165	min : 67 max : 67 avg : 67	min : 0.3 max : 71.0 avg : 4.1	min : 85,904 max : 420,912 avg : 100,100	100%
Collberg-0004-2	min : 408 max : 408 avg : 408	min : 11,294 max : 2,999,784 avg : 138,738	min : 332 max : 332 avg : 332	min : 219 max : 219 avg : 219	min : 722 max : 4,765 avg : 1,400	min : 128 max : 128 avg : 128	min : 1.6 max : 275.2 avg : 13.2	min : 172,760 max : 1,243,348 avg : 206,449	100%
Collberg-0004-3	min : 203 max : 203 avg : 203	min : 5,503 max : 1,439,344 avg : 96,331	min : 78 max : 78 avg : 78	min : 171 max : 171 avg : 171	min : 718 max : 3,478 avg : 1,317	min : 78 max : 78 avg : 78	min : 0.5 max : 138.9 avg : 11.1	min : 86,948 max : 755,056 avg : 137,375	100%
Collberg-0004-4	min : 307 max : 307 avg : 307	min : 8,674 max : 9,279,883 avg : 533,675	min : 146 max : 146 avg : 146	min : 274 max : 274 avg : 274	min : 725 max : 5,424 avg : 1,452	min : 146 max : 146 avg : 146	min : 0.7 max : 1,681.6 avg : 86.6	min : 103,964 max : 7,480,952 avg : 482,005	100%
FNV1a	min : 143 max : 143 avg : 143	min : 1,499 max : 54,846 avg : 3,544	min : 57 max : 57 avg : 57	min : 110 max : 110 avg : 110	min : 517 max : 1,180 avg : 861	min : 57 max : 57 avg : 57	min : 0.1 max : 4.9 avg : 0.3	min : 80,872 max : 101,828 avg : 82,139	100%
Jenkins	min : 201 max : 201 avg : 201	min : 5,520 max : 1,069,111 avg : 76,420	min : 125 max : 125 avg : 125	min : 79 max : 79 avg : 79	min : 631 max : 1,888 avg : 1,076	min : 125 max : 125 avg : 125	min : 0.5 max : 83.9 avg : 6.2	min : 87,572 max : 543,272 avg : 110,694	100%
JodyHash	min : 92 max : 92 avg : 92	min : 1,349 max : 155,637 avg : 9,820	min : 48 max : 48 avg : 48	min : 90 max : 90 avg : 90	min : 468 max : 1,085 avg : 803	min : 48 max : 48 avg : 48	min : 0.1 max : 25.2 avg : 1.4	min : 79,732 max : 203,072 avg : 86,237	100%
MD5	min : 9,743 max : 9,743 avg : 9,743	min : 173,673 max : 47,927,795 avg : 2,328,114	min : 557 max : 557 avg : 557	min : 314 max : 314 avg : 314	min : 1,311 max : 4,828 avg : 1,857	min : 557 max : 557 avg : 557	min : 16.5 max : 4,226.7 avg : 207.5	min : 266,032 max : 2,688,976 avg : 583,198	100%
SpiHash	min : 364 max : 364 avg : 364	min : 2,880 max : 1,694,015 avg : 100,661	min : 160 max : 160 avg : 160	min : 362 max : 362 avg : 362	min : 824 max : 1,829 avg : 1,224	min : 160 max : 160 avg : 160	min : 0.3 max : 288.1 avg : 15.1	min : 89,356 max : 1,434,764 avg : 159,257	100%
SpookyHash	min : 536 max : 536 avg : 536	min : 1,784 max : 140,565 avg : 9,571	min : 79 max : 79 avg : 79	min : 426 max : 426 avg : 426	min : 788 max : 1,443 avg : 1,125	min : 79 max : 79 avg : 79	min : 0.1 max : 23.1 avg : 1.3	min : 82,424 max : 193,080 avg : 88,364	100%
SuperFastHash	min : 182 max : 182 avg : 182	min : 1,402 max : 37,479 avg : 3,502	min : 81 max : 81 avg : 81	min : 144 max : 144 avg : 144	min : 506 max : 1,331 avg : 874	min : 81 max : 81 avg : 81	min : 0.1 max : 3.1 avg : 0.3	min : 82,572 max : 94,696 avg : 83,540	100%
Xxhash	min : 186 max : 186 avg : 186	min : 1,672 max : 103,193 avg : 9,310	min : 68 max : 68 avg : 68	min : 182 max : 182 avg : 182	min : 691 max : 1,470 avg : 1,047	min : 68 max : 68 avg : 68	min : 0.1 max : 16.3 avg : 1.1	min : 83,376 max : 164,128 avg : 88,478	100%

TABLE 5.2 – Moyenne de toutes les protections par fonction de hachage

un ensemble de nombres aléatoires (un nombre d'entiers prédéfinis et un nombre d'entiers aléatoires) et nous vérifions que les deux binaires renvoient les mêmes résultats en sortie. Si pour toutes les entrées nous avons les mêmes sorties, alors nous considérons que la sémantique est restée la même lors de la dévirtualisation.

Concernant la concision, nous prenons plusieurs métriques comme le nombre d'instruction avant l'application des protections, après l'application des protections et après la dévirtualisation. Le résultat de ces métriques peut être consulté dans le Tableau 5.3 et plus en détail dans le Tableau 5.2 (colonnes *Binary* et *Trace Size*).

Résultats : Le Tableau 5.4 donne une moyenne des ratios (en termes du nombre d'instructions) du binaire d'origine vers le binaire virtualisé et ensuite du binaire d'origine vers celui dévirtualisé. Cette table nous montre (a) qu'après avoir appliqué notre approche, nous sommes en mesure de reconstruire un binaire valide (en termes de correction) pour 100% de nos échantillons, (b) qu'après avoir appliqué les protections, la taille des binaires ainsi que la taille des traces sont considérablement augmentées mais après avoir appliqué notre approche, nous sommes en mesure de reconstruire des binaires plus petits que ceux d'origine. Cet effet est dû au fait que nous concrétisons tout ce qui n'est pas en lien avec l'entrée utilisateur.

	Original	Obfuscated	Deobfuscated
Binary Size	min : 78 max : 426 avg : 196	min : 468 max : 5,424 avg : 1,205	min : 48 max : 557 avg : 119
Trace Size	min : 92 max : 9,743 avg : 726	min : 1,349 max : 47,927,795 avg : 229,168	min : 48 max : 557 avg : 143

Table 5.3 – Taille de nos échantillons

	Original → Obfuscate	Original → Deobfuscate
Correctness	100% (920 successes)	
Binary Size	min : x3.3 max : x14.0 avg : x6	min : x0.1 max : x2.8 avg : x0.71
Trace Size	min : x17 max : x1252 avg : x424	min : x0.05 max : x0.9 avg : x0.39

Table 5.4 – Moyenne des ratios sur nos 920 échantillons

Conclusion : Notre approche conserve les mêmes propriétés fonctionnelles lors de la phase de dévirtualisation et compte tenu des résultats des métriques nous sommes en mesure de réduire la taille des binaires de manière conséquente, voire proche de celle d'origine (et parfois même plus petite).

5.4.2 Efficacité (C_2)

Objectif : Le critère C_2 vise à déterminer l'efficacité de l'approche en termes de temps d'exécution et de passage à l'échelle.

Métriques utilisées : Pour répondre à cette question nous prenons plusieurs mesures à chaque étape de l'analyse ainsi que toutes les 10,000 instructions traitées. Les résultats de ces métriques sont présentés en détail dans le Tableau 5.2 (colonnes *Obfuscated (trace size)* et *Time*).

Résultats : La Figure 5.13 représente les paliers en temps absolu de chaque dévirtualisation de nos 920 échantillons. Comme on peut le constater, c'est 80% de nos échantillons que nous pouvons dévirtualiser en moins de 5 secondes. Le temps le plus long qu'il a fallu pour dévirtualiser un binaire est de 4,226 seconds (∼1h10) pour pas loin de 48 millions d'instructions traitées [12].

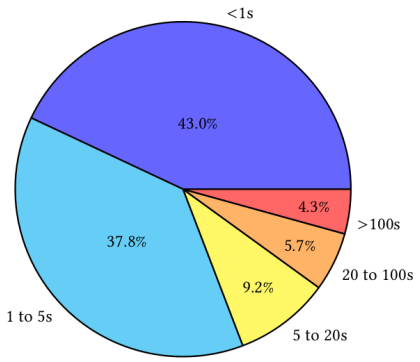

FIGURE 5.13 – Palier de temps pour nos 920 échantillons

Si nous zoomons sur le temps d'analyse nécessaire pour dévirtualiser toutes

12. Fonction de hachage MD5 avec deux niveaux de virtualisation (nested vm=2).

les protections appliquées à la fonction de hachage MD5 [13], nous obtenons la Figure 5.14. Chacune des 46 courbes pointillées représente une protection. Comme nous pouvons le constater, le temps d'analyse est proportionnel au nombre d'instructions exécutées. Nous pouvons également constater qu'il existe une variation du temps d'exécution entre les protections pour un même nombre d'instructions exécutées. Cette variation est due au fait que certaines instructions sont plus ou moins longues à décoder et à représenter.

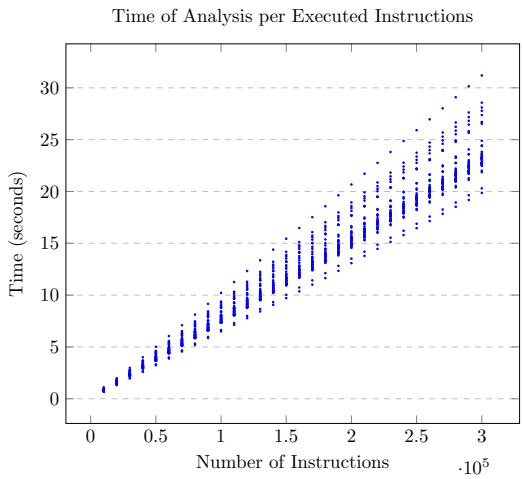

FIGURE 5.14 – Temps d'analyse pour toutes les protections de la fonction de hachage MD5

Conclusion : Dans nos expérimentations, on constate que notre approche possède un temps d'analyse proportionnel au nombre d'instructions exécutées. Plus les protections rajoutent des instructions dans le binaire protégé, plus le temps d'analyse augmente de façon proportionnelle – la complexité des expressions devrait généralement jouer un rôle, ici les contraintes sont simples à résoudre. Compte tenu de nos résultats, notre approche nous a permis de dévirtualiser la plupart de nos échantillons en quelques secondes et quelques minutes pour les échantillons les plus compliqués tels que MD5 (*est-ce que 1h10 d'analyse est aberrant pour une personne ou une entité voulant vraiment casser la protection ?*)

13. À des fins représentatives, nous choisissons MD5 car c'est la fonction de hachage qui utilise le plus d'instructions.

5.4.3 Influence des Protections (C_3)

Objectif : Le critère C_3 vise à déterminer s'il y a des protections qui influencent la correction, la concision ainsi que les performances de notre approche. Si oui, à quel point ?

Métriques utilisées : Pour répondre à ce critère nous regardons les différences en termes d'instructions de chaque binaire dévirtualisé sur 46 protections pour un même échantillon. S'il existe des différences, c'est que la protection influence notre approche, dans le cas contraire on peut en déduire que la protection n'a pas d'impact sur le processus de dévirtualisation.

Résultats : Si nous regardons les résultats des métriques dans le Tableau 5.2 (colonnes *Deobfuscated*), nous pouvons dire que la concision de notre approche est la même quelle que soit la protection appliquée. Les protections n'influencent donc pas notre processus de dévirtualisation et cela est vrai pour les 20 échantillons analysés.

Si nous isolons les résultats de ces métriques sur les dispatcheurs utilisés pour protéger le calcul MD5, nous obtenons le diagramme représenté par la Figure 5.15. Nous pouvons constater que le nombre d'instructions générées par la protection fluctue en fonction du dispatcheur utilisé mais que le résultat après dévirtualisation reste identique quelque soit la dispatcheur utilisé. Cependant, le nombre d'instructions générées par la protection influence le temps d'exécution (Voir C_2) et cela peut poser des problèmes non négligeables sur l'analyse de grosses fonctions. Par exemple, toujours avec l'échantillon de MD5, le surcout d'exécution est de x10 pour un seul niveau de virtualisation, x100 pour deux niveaux de virtualisation et x6800 pour un troisième niveau.

Conclusion : Notre approche n'est pas influencée par les 46 protections de Tigress utilisées et les résultats des 46 binaires dévirtualisés issues d'un même échantillon sont identiques. Si de telles protections ne rentrent pas en interaction avec nos variables symboliques (les paramètres de la fonction virtualisés), elles n'ont pas d'impact sur la concision (critère C_1) de notre analyse. En revanche les résultats précédent (C_2) démontrent que toutes les protections considérées ont un effet sur l'efficacité directement proportionnel à l'augmentation qu'elles impliquent sur la taille des traces.

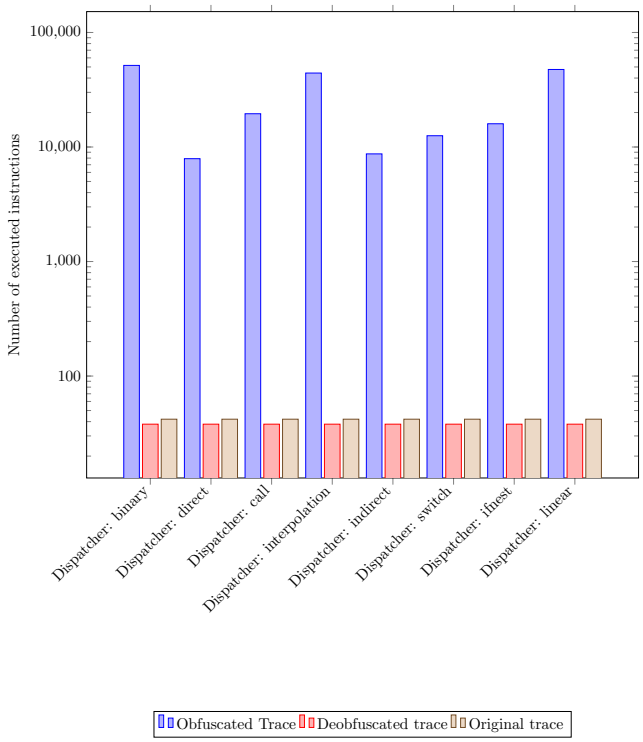

FIGURE 5.15 – Influence des dispatcheurs sur notre analyse

5.5 Deuxième expérimentation : Le challenge Tigress

Nous avons choisi le challenge Tigress comme étude de cas pour deux raisons (a) certaines machines virtuelles du challenge n'ont pas été résolues, (b) cela nous permet de voir si notre approche fonctionne sur un environnement non contrôlé avec plusieurs combinaisons de protection. Le Challenge[14] est composé est 35 machines virtuelles avec différents niveaux de protection (voir Tableau 5.5).

Tous les challenges ont la même structure : les binaires possèdent une fonction virtualisée qui effectue une transformation sur une entrée donnée et renvoie le nombre dérivé (voir Listing 5.9). Le but est de trouver l'algorithme de hachage sous la forme de pseudo code (chaque algorithme est propriétaire) le plus proche possible de celui d'origine. Dans notre cas, au lieu de renvoyer un pseudo code nous renvoyons la représentation intermédiaire d'LLVM pour compiler l'algorithme de

14. `http://tigress.cs.arizona.edu/challenges.html#current`

Challenge	Description	Difficulty	Web Status	Our Status
0000	One level of virtualization, random dispatch.	1	Solved	Solved
0001	One level of virtualization, superoperators, split instruction handlers.	2	Open	**Solved**
0002	One level of virtualization, bogus functions, implicit flow.	3	Open	**Solved**
0003	One level of virtualization, instruction handlers obfuscated with arithmetic encoding, virtualized function is split and the split parts merged.	2	Open	**Solved**
0004	Two levels of virtualization, implicit flow.	4	Open	**Solved**
0005	One level of virtualization, one level of jitting, implicit flow.	4	Open	Open†
0006	Two levels of jitting, implicit flow.	4	Open	Open†

† : JIT non supporté par notre outil.

Table 5.5 – Challenge Tigress
(chaque challenge contient 5 machines virtuelles)

hachage en une version binaire non virtualisée.

Sur les 35 machines virtuelles nous nous sommes concentrés sur les 25 premières (de `0000` à `0004`) et nous n'avons pas analysé les VM utilisant le JIT comme protection (`0005` et `0006`). Dans ces derniers challenges, la protection dite "JIT" (*Just In Time*) est le fait de traduire le *bytecode* de la machine virtuelle en *opcode* exécutables par la machine hôte. Ce mécanisme n'étant pas supporté par notre outil, nous avons omis ces challenges.

Pour les challenges concernant uniquement la virtualisation, 15 VMs on été dévirtualisées avec une seule exécution car ces dernières n'avaient pas de chemin dépendant de l'entrée utilisateur. Puis en appliquant une couverture de chemins (étape 3), il nous a été possible de dévirtualiser les 10 VMs restantes.

	Tigress challenges				
	VM-0	**VM-1**	**VM-2**	**VM-3**	**VM-4**
0000	3.85s	9.20s	3.27s	4.26s	1.58s
0001	1.26s	1.42s	3.27s	2.49s	1.74s
0002	6.58s	2.02s	2.63s	4.85s	3.82s
0003	45.6s	11.3s	8.84s	4.84s	21.6s
0004	361s	315s	588s	8049s	1680s

Table 5.6 – Temps (en secondes) pour résoudre les VMs du challenge

Le Tableau 5.6 illustre le temps nécessaire pour résoudre les challenges. Notons que la consommation mémoire est proportionnelle au temps d'analyse (voir Section 5.4.2). Comme on peut le constater c'est le challenge `0004` qui prend le plus de temps. Ce challenge possède deux niveaux de virtualisation et c'est pas loin de 140 millions d'expressions qui ont été traitées et simplifiées en 320 en seulement 2

	Tigress challenges				
	VM-0	**VM-1**	**VM-2**	**VM-3**	**VM-4**
0000	x0.85	x1.09	x0.73	x0.89	x1.4
0001	x0.41	x0.60	x0.26	x0.22	x0.53
0002	x0.29	x0.28	x0.51	x1.40	x0.42
0003	x1.10	x1.17	x1.57	x0.46	x0.44
0004	x0.81	x0.38	x0.70	x0.37	x0.53

TABLE 5.7 – Ratio (taille) original → dévirtualisé

heures d'analyse. Le Tableau 5.7 illustre le ratio en termes de taille entre la version originale et la version dévirtualisée [15].

5.6 Discussion sur les protections attaquées

Cette section est une discussion ouverte sur le rôle que joue chaque protection et pourquoi elles n'impactent pas la concision de notre approche.

Opaque to VPC, Random Opcodes, Dispatchers, Anti Branch Analysis, Bogus Function and Loop Iterations, Encode Byte Array, Super Operators : Ces protections on principalement été conçues pour ralentir les analyses statiques mais en utilisant une approche dynamique et notre approche, nous sommes en mesure de distinguer les instructions qui font partie du fonctionnement de la machine virtuelle et celles qui sont utilisées pour simuler le comportement du programme d'origine. En isolant ces dernières instructions et en concrétisant toutes les autres (voir Section 5.2.4), il nous est possible de reconstruire un programme sans virtualisation (voir Section 5.2.5 et 5.2.6).

Kind of Operands and Duplicate Opcodes : Tigress fournit deux mécanismes de gestion d'opérandes pour leur ISA. L'un avec les opérandes placées sur la pile et l'autre avec les opérandes placées dans les registres (l'option *Duplicate Opcodes* mixe les deux). Dans la représentation intermédiaire de Triton, il n'y a pas de distinction entre une lecture venant de la mémoire et une lecture venant des registres (pareil pour les écritures). Ainsi, et par effet de bord, ces deux mécanismes sur la

15. Afin de pouvoir effectuer ces ratios, nous avons demandé à Christian Collberg les sources des ses challenges une fois résolue.

gestion des opérandes sont traités de la même façon et n'ont donc pas d'impact sur le résultat de notre analyse.

Nested VM : Cette protection n'influence pas la concision de notre approche cependant elle a un réel impact sur le temps d'analyse. Cette protection ajoute de façon drastique un grand nombre d'instructions à chaque niveau de virtualisation. Sur notre machine d'expérimentation (voir Section 5.3.3) nous ne sommes capables de résoudre que deux niveaux de virtualisation. Cependant, nous avons réussi à résoudre trois niveaux de virtualisation en utilisant le Cloud d'Amazon (EC2 [16]). Cela signifie que cette protection impacte seulement les moyens matériel mis à disposition pour la casser.

5.7 Limitations et contre-mesures

Dans cette section nous discutons des limitations de notre approche ainsi que des contre-mesures pouvant être misent en place afin de contrer notre analyse. Pour cela, nous commençons par introduire les définitions des termes résultat correct, résultat complet et résultat pertinent.

5.7.1 Propriétés attendues

Soit la fonction f' obtenue à partir d'une fonction f avec notre approche. On note ϕ_w la disjonction logique de l'ensemble des prédicats de chemin calculé en reconstruisant f'. On définit $S_{f'}$ comme l'ensemble des valeurs satisfaisant ϕ_w. $S_{f'}$ est le support de construction de f'.

Résultat correct : Soit f' la fonction reconstruite d'une fonction virtualisée f, et $S_{f'}$ son support de construction. On considère que la fonction f' est *correcte* si pour toutes les entrées dans $S_{f'}$, la valeur retournée par l'exécution de f' est identique à celle retournée par l'exécution de f. Soit : $\forall x \in S_{f'}$, $f'(x) = f(x)$. Si la fonction dévirtualisée est correcte, on dit que le résultat de notre approche est correct.

Résultat complet : Soit f' la fonction reconstruite d'une fonction virtualisée f, et $S_{f'}$ son support de construction comprenant toutes les entrées ($x \in X$).

16. `https://aws.amazon.com/ec2/instance-types/`

On considère que la fonction f' est *complète* si pour toutes les entrées dans $S_{f'}$, la valeur retournée par l'exécution de f' est identique à celle retournée par l'exécution de f. Soit : $\forall x \in S_{f'}$, $f'(x) = f(x)$. Si la fonction dévirtualisée est complète, on dit que le résultat de notre approche est complet.

Résultat pertinent : Soit f' la fonction reconstruite d'une fonction virtualisée f. On dit que le résultat de notre analyse est pertinent si la concision (critère C_1, Section 5.4.1) de la fonction $f'(x)$ est bénéfique pour un analyste.

Le Tableau 5.8 est l'aperçu des conséquences de chacune des limitations sur le résultat de notre approche, ces limitations étant ensuite discutées dans les sections qui suivent.

	Résultat de notre approche
Présence d'index symbolique	incorrect & incomplet
Couverture incomplète des chemins	incomplet
Timeout du solveur de contraintes	incomplet
sous-approximation de la teinte	incorrect & incomplet
sur-approximation de la teinte	non pertinent
Bytecode protégé	non pertinent
†Cas nominal	correct, complet et pertinent

Table 5.8 – Aperçu des conséquences de chacune des limitations sur le résultat de notre approche.
† : indexes concrets, peu de chemins, teinte et solveur ok

5.7.2 Accès mémoire à index symbolique

La politique de concrétisation du moteur symbolique de Triton (voir Section 3.5.3, Figure 3.10) ne nous permet pas d'avoir une représentation des accès mémoires sous la forme symbolique. Cela signifie que l'implémentation de notre approche (voir Section 5.2.7) ne nous permet de reconstruire des accès mémoire dépendants des entrées de la fonction (arguments teintés).

```
01. char bytes[] = {
02.   0x11, 0x22, 0x33, 0x44, 0x55, 0x66,
03.   0x77, 0x88, 0x99, 0xaa, 0xbb, 0xcc,
04.   0xdd, 0xee, 0xff
05. };
06.
```

```
07. unsigned f(unsigned x) {
08.   int hash = 1;
09.   while (x) {
10.     hash *= bytes[(x & 0xff) % sizeof(bytes)];
11.     x >>= 1;
12.   }
13.   return hash;
14. }
```

Listing 5.10 – Exemple d'index symbolique

Le Listing 5.10 illustre un exemple de structure de code impliquant un index symbolique. Dans cet exemple, l'argument x de la fonction f est symbolique ainsi que teinté. À la ligne 10, x est utilisé comme index dans le tableau *bytes* et le contenu de ce tableau est utilisé dans le calcul du *hash*. Compte tenu de la politique de concrétisation de Triton, ce lien entre l'argument de la fonction (qui est symbolique) et l'indexation du tableau dans le calcul du hash est perdu (concrétisé pour une entrée donnée). Cela signifie que le résultat de notre approche sera correct uniquement pour une seule entrée (celle courante) mais incorrect pour le reste des valeurs possibles de x et par conséquent incomplet.

Conclusion : En présence d'indexation symbolique, le résultat de notre approche est incorrect et incomplet.

5.7.3 Couverture de chemins et timeout

Explosion combinatoire : La complétude de notre approche est basée sur le fait de pouvoir lister l'ensemble des chemins d'une fonctions virtualisée. Cela pose problème sur des grosses fonctions en raison de l'explosion combinatoire qu'implique l'exploration des chemins.

Conclusion : Si l'ensemble des chemins n'est pas énuméré, le résultat de notre approche est incomplet.

Expressions trop complexes : Comment mentionné précédemment, nous devons identifier toutes les branches possibles d'une fonction afin d'avoir un résultat complet. Pour cela, nous faisons appel au solveur de contraintes pour résoudre chaque condition de branchement et ainsi parcourir toutes les branches (voir Section 5.2.5.1) de la fonction virtualisée. Une contre-mesure possible pour cette étape

serait d'intégrer des expressions complexes à résoudre pour chaque condition de branchement (conditions dépendantes des entrées teintées) afin d'engendrer un délai non négligeable (ex : *timeout*) côté solveur de contraintes. Ci-dessous un exemple avec le Listing 5.11 où $hash(x)$ serait une fonction de hachage cryptographique non réversible.

```
int f(int x) {
    if (hash(x) == 0x1122334455667788)
        /* ... */
    else
        /* ... */
}
```

Listing 5.11 – Contre-mesure possible pour notre étape de couverture de chemin

Conclusion : En présence de timeout levé par le solveur de contraintes, le résultat de notre approche est incomplet.

5.7.4 Sous ou sur-approximation de la teinte

Étant donné que nous nous reposons sur une analyse de teinte pour dissocier les instructions qui font partie de la machine virtuelle de celles qui sont exécutées pour simuler le programme d'origine, il serait possible pour un défenseur d'entrelacer ces deux séquences d'instructions pour (1) engendrer une sous-approximation de la teinte afin d'omettre certaines instructions utilisées dans le calcul du programme d'origine et donc d'impliquer une incorrection sur le résultat final, (2) engendrer une sur-approximation de la teinte afin d'intégrer le plus d'instructions teintées possibles sur la trace d'exécution. Par exemple, si nous imaginons que toutes les instructions exécutées faisant partie de la machine virtuelle sont teintées, le résultat de notre approche serait la machine virtuelle dans son intégralité (ce qui ne serait pas pertinent pour un analyste).

Conclusion : En cas d'une sous-approximation de la teinte, le résultat de notre approche est incorrect et incomplet. En cas d'une sur-approximation le résultat de notre approche est non pertinent.

5.7.5 Protéger le bytecode plutôt que la VM

Une contre-mesure possible contre des attaques statiques et dynamiques serait de protéger le bytecode de la machine virtuelle plutôt que ses composants (l'un n'empêchant pas l'autre). Prenons comme exemple le code illustré dans le Listing 5.12. Cette fonction f effectue une simple multiplication de ses deux arguments.

```
int f(int x, int y) {
  return x * y;
}
```

Listing 5.12 – Échantillon de fonction à virtualiser

Supposons maintenant que nous voulions protéger ce calcul en appliquant une protection par virtualisation. Le Listing 5.13 illustre un exemple de bytecode possible qui sera simulé par une machine virtuelle. Dans notre exemple, le registre *r9* est l'argument x et le registre *r10* l'argument y. C'est deux registres sont mis respectivement dans *r0* et *r1*, puis une multiplication est effectuée en plaçant le résultat dans *r0*. Le registre *r0* étant le registre de retour (comme *rax* pour x86-64).

```
31 01 00 09 : MOV r0, r9
31 01 01 0a : MOV r1, r10
44 00 00 01 : MUL r0, r0, r1
60          : RET
```

Listing 5.13 – Bytecode de la fonction f

Nos expérimentations (voir Section 5.3) montrent clairement que ce genre de code virtualisé (simple fonction avec un calcul sans effet de bord) permet très facilement de retrouver son code d'origine ($x \times y$). En protégeant le bytecode de la machine virtuelle par le biais de passes d'obfuscation, cela rendrait le résultat de notre approche moins pertinent. Par exemple, l'utilisation de MBA rendrait la compréhension des expressions arithmétiques et logiques après dévirtualisation plus compliquées. Par exemple, Yongxin Zhou *et al.* [101] montrent qu'une opération $x \times y$ peut être transformée en une expression $(x \wedge y) \times (x \vee y) + (x \wedge (\neg y)) \times (\neg x \wedge y)$. Si cette transformation est appliquée sur le bytecode du Listing 5.13, notre approche ne pourrait pas retrouver le code d'origine $x \times y$ et aura pour résultat l'expression issue du MBA, ce qui nuirait à la compréhension de la fonction f.

Conclusion : En cas de bytecode protégé, le résultat de notre approche est non pertinent.

5.8 Travaux similaires

Nous discutons ici des travaux similaires sur la dévirtualisation tout en positionnant notre approche.

5.8.1 Dévirtualisation manuelle et heuristiques

Sharif *et al.* [88] proposent une approche dynamique qui vise à identifier le VPC basé sur la reconnaissance de schémas connus des accès mémoire et arrivent ensuite à reconstruire un CFG. Leur approche souffre cependant de certaines limitations. Par exemple, leur stratégie de détection de boucles n'est pas applicable à une machine virtuelle qui utiliserait des *threads* dans ses composants (ex : le dispatcher). Un autre point est que leur approche considère que chaque cellule mémoire se voit affecter une unique variable *abstraite*, ce qui signifie qu'un attaquant pourrait utiliser le même emplacement mémoire pour différentes variables à des moments différents de l'exécution pour introduire une imprécision dans leur analyse. Inversement, notre approche devrait résoudre ce problème dû fait que nous travaillons sur des traces d'exécution et que la concrétisation des accès mémoire fournit une gestion de l'*aliasing* assez gratuite. Ils mentionnent également un problème lors de l'analyse de machines virtuelles contenant plusieurs niveaux de virtualisation. Cette récursion de virtualisation est un facteur de limitation pour leur approche. Ce problème est quelque chose que nous prenons en compte au travers des VMs `0004` du Tigress challenge ainsi que dans notre environnement contrôlé et que nous arrivons à dévirtualiser.

5.8.2 Dévirtualisation basée sur la sémantique

Coogan *et al.* [32] proposent une approche qui vise à se concentrer sur la détection des instructions qui ont un effet observable sur la comportement du programme virtualisé. Ils commencent par utiliser un traceur afin de récupérer toutes les informations bas niveau de l'instruction courante exécutée telles que les

opcodes, mnémoniques, état des registres et accès mémoire. Ils monitorent ensuite les appels système ainsi que leurs arguments en se basant sur une base de données qui donne l'interface binaire-programme (ABI). Basé sur ces informations, ils essaient de comprendre quelles sont les instructions qui affectent les arguments des appels système et introduisent le termes de *instructions pertinentes*. À la fin ils construisent une sous-trace depuis ces instructions pertinentes. Cette approche est très proche de la notre, cependant, elle ne peut dévirtualiser qu'un seul chemin et ne fonctionne pas si la partie virtualisée ne contient pas d'appels système. À l'inverse de leur approche, nous utilisons une analyse de teinte dite *en avant* (*forward taint analysis*) pour isoler ces instructions pertinentes et nous utilisons ensuite une représentation symbolique de ces instructions afin d'effectuer une couverture de chemins, et donc, retrouver l'arbre d'exécution du programme virtualisé. Cet arbre d'exécution est ensuite compilé en une nouvelle version binaire non virtualisée.

Yadegari *et al.* [99] proposent une approche générique de dévirtualisation qui combine analyse de teinte, exécution symbolique et passes de simplification. Leur but est de trouver le graphe de flot de contrôle d'un malware et ils effectuent des expérimentations avec plusieurs outils de virtualisation afin d'évaluer leur approche. Nos travaux montrent des similarités avec leur méthode. Toutefois, nous considérons le problème de récupérer un code binaire sémantiquement correct (et non virtualisé) afin d'en comprendre son fonctionnement opérationnel et donc de récupérer la propriété intellectuelle. Nous effectuons également un grand nombre d'expériences contrôlées en combinant les différentes options de virtualisation fournies par l'outil Tigress afin d'évaluer les propriétés de notre approche.

Kinder [60] propose une méthode reposant sur l'analyse statique et l'interprétation abstraite. Son approche définit un domaine abstrait intitulé *vpc-sensitve* qui est construit depuis une analyse de valeur effectuée sur toute la machine virtuelle. Son but est de fournir à l'analyste des invariants sur le code (dans son cas il donne le nombre d'arguments ainsi leur valeur pour chaque appel externe). La seule contrainte pour que l'approche fonctionne est de connaitre la position du VPC à chaque point de programme. De plus, son approche ne propose pas de solution complète pour analyser des programmes virtualisés de façon générique mais se concentre plutôt sur les effets particuliers de la virtualisation.

5.8.3 Exécution symbolique

Banescu *et al.* [15] ont récemment évalué l'efficacité des mécanismes d'obfuscation standard contre la déobfuscation symbolique. Ils concluent, comme nous, que sans défense visant précisément l'analyse symbolique, la plupart des protections ne sont pas efficaces. Nos travaux sont complémentaires de [15] car nous nous concentrons uniquement sur la protection basée sur la virtualisation mais nous la couvrons d'une manière un peu plus poussée et nous prenons une notion plus ambitieuse de déobfuscation (récupérer un code équivalent et petit) tout en considérant la couverture du programme.

5.9 Conclusion

La protection des logiciels basée sur la virtualisation a pris une place importante au cours de la dernière décennie afin de protéger les logiciels légitimes ainsi que les logiciels malveillants. En parallèle, les chercheurs ont publié plusieurs approches pour analyser ces protections. Pourtant, alors que l'objectif ultime de la déobfuscation est de récupérer le programme original, ces approches se concentrent plutôt sur des étapes intermédiaires, telles que la récupération des composants de la machine virtuelle ou la simplification des traces.

En étudiant l'état de l'art, on peut trouver trois types d'approches :

1. L'analyse semi-manuelle qui implique une rétro-ingénierie des handlers, puis l'écriture d'un désassembleur de l'ISA de la machine virtuelle. De telles approches prennent du temps ;
2. L'analyse statique automatisée (ex. interprétation abstraite) qui vise à récupérer le comportement du programme d'origine et son CFG. De telles approches ne fonctionnent pas si l'architecture de la machine virtuelle est, elle-même, protégée ou si elle utilise plusieurs niveaux de virtualisation ;
3. L'analyse dynamique automatisée (ex. exécution symbolique) qui vise à récupérer le comportement du programme d'origine et son CFG, mais cette fois, en exécutant le binaire protégé. De telles approches sont puissantes contre les machines virtuelles mais ne se concentrent généralement que sur l'analyse d'un seul chemin.

Dans ce chapitre, nous proposons une nouvelle approche dynamique et automatisée qui vise à récupérer le comportement d'origine d'un code virtualisé. Nous

démontrons le potentiel de la méthode grâce à une évaluation expérimentale approfondie, en évaluant sa précision, son efficacité et sa généricité et nous résolvons les challenges (excepté la partie JIT) du défi Tigress dans un manière complètement automatisée.

D'un point de vue des expérimentations, ce travail démontre clairement que les fonctions de hachage (typiques des ressources propriétaires protégées par l'obfuscation) peuvent être facilement récupérées à partir de leurs versions virtualisées. Cela met en cause l'utilisation de la virtualisation comme protection contre la rétro-ingénierie à moins que les défenseurs ne soient prêts à en payer le temps d'exécution en utilisant des virtualisations imbriquées. Par conséquent, les défenseurs doivent prendre grand soin de protéger la machine virtuelle ainsi que son bytecode contre les attaques dynamiques. Bien que notre approche montre encore des limites sur les classes de programmes qui peuvent être gérés, nous nous concentrerons, dans un avenir proche, sur la reconstruction de structures de programme plus complexes telles que des boucles et des accès mémoire dépendant de l'utilisateur.

5.10 Annexes

	Traces Size (instructions)			Binary Size (instructions)		
Protection	**Original**	**Obfuscated**	**Deobfuscated**	**Original**	**Obfuscated**	**Deobfuscated**
Anti Branch Analysis : branchFuns	min : 92 max : 9,743 avg : 698	min : 3,047 max : 1,555,703 avg : 121,460	min : 48 max : 599 avg : 122	min : 78 max : 426 avg : 192	min : 935 max : 2,048 avg : 1,641	min : 48 max : 599 avg : 122
Kind of Operands : stack	min : 92 max : 9,743 avg : 698	min : 1,430 max : 381,230 avg : 31,104	min : 48 max : 599 avg : 122	min : 78 max : 426 avg : 192	min : 783 max : 1,139 avg : 979	min : 48 max : 599 avg : 122
Kind of Operands : registers	min : 92 max : 9,743 avg : 698	min : 1,459 max : 425,285 avg : 34,322	min : 48 max : 599 avg : 122	min : 78 max : 426 avg : 192	min : 807 max : 1,182 avg : 1,065	min : 48 max : 599 avg : 122
Opaque to VPC : False	min : 92 max : 9,743 avg : 698	min : 1,430 max : 381,230 avg : 31,104	min : 48 max : 599 avg : 122	min : 78 max : 426 avg : 192	min : 783 max : 1,139 avg : 979	min : 48 max : 599 avg : 122
Opaque to VPC : True	min : 92 max : 9,743 avg : 698	min : 1,600 max : 700,138 avg : 51,405	min : 48 max : 599 avg : 122	min : 78 max : 426 avg : 192	min : 861 max : 1,296 avg : 1,148	min : 48 max : 599 avg : 122
Duplicate Opcodes : 3	min : 92 max : 9,743 avg : 698	min : 1,430 max : 381,230 avg : 31,037	min : 48 max : 599 avg : 122	min : 78 max : 426 avg : 192	min : 783 max : 1,226 avg : 1,063	min : 48 max : 599 avg : 122
Dispatcher : binary	min : 92 max : 9,743 avg : 698	min : 2,449 max : 2,825,359 avg : 195,969	min : 48 max : 599 avg : 122	min : 78 max : 426 avg : 192	min : 814 max : 1,154 avg : 1,010	min : 48 max : 599 avg : 122
Dispatcher : interpolation	min : 92 max : 9,743 avg : 698	min : 2,625 max : 2,592,186 avg : 181,698	min : 48 max : 599 avg : 122	min : 78 max : 426 avg : 192	min : 839 max : 1,183 avg : 1,037	min : 48 max : 599 avg : 122
Dispatcher : linear	min : 92 max : 9,743 avg : 698	min : 2,115 max : 5,804,970 avg : 351,747	min : 48 max : 599 avg : 122	min : 78 max : 426 avg : 192	min : 785 max : 1,125 avg : 982	min : 48 max : 599 avg : 122
Nested VMs : 1	min : 92 max : 9,743 avg : 698	min : 1,430 max : 381,230 avg : 30,104	min : 48 max : 599 avg : 122	min : 78 max : 426 avg : 192	min : 783 max : 1,139 avg : 979	min : 48 max : 599 avg : 122
Nested VMs : 2	min : 92 max : 9,743 avg : 698	min : 37,479 max : 47,927,795 avg : 3,520,624	min : 48 max : 599 avg : 122	min : 78 max : 426 avg : 192	min : 676 max : 1,182 avg : 814	min : 48 max : 599 avg : 122

Table 9 – Moyenne de toutes les fonctions de hachage par protection

Chapitre 6

Conclusion et ouverture

Cette thèse s'inscrit dans la problématique de l'automatisation de la rétro-ingéniérie et a été réalisée dans un cadre industriel au sein de Quarkslab avec l'objectif de rester proche des missions afin de comprendre les problématiques auxquelles les analystes sont confrontés et en leur proposant des solutions. Certaines de ces problématiques, ainsi que celles issues d'un sondage de différentes entités, sont présentées dans le Chapitre 2. Le framework Triton a été développé en s'appuyant sur les retours d'expériences des analystes, ainsi que sur mon expérience personnelle dans la rétro-ingénierie. Triton est un framework permettant d'appliquer une analyse de teinte ainsi qu'une exécution symbolique sur du code binaire. Ces composants, ainsi que des optimisations y sont présentés et formalisés dans le Chapitre 3. Dans le Chapitre 4, nous présentons un cas réel d'utilisation de Triton pour la détection de prédicats opaques dans les conditions de branchement au sein du malware X-Tunnel et comparons nos résultats avec une étude existante. Enfin, dans le Chapitre 5, nous présentons une nouvelle approche dynamique et automatique permettant la dévirtualisation de code binaire en combinant une analyse de teinte, une exécution symbolique dynamique et une politique de simplification de code. Nous discutons également des limitations et des garanties de notre approche puis nous démontrons le potentiel de cette dernière en résolvant automatiquement une partie du challenge Tigress.

Contributions : Cette thèse a permis de mieux comprendre les problématiques des industriels en rétro-ingénierie et de mieux appréhender les possibilités de l'outillage à la résolution de ces derniers. Les outils occupent une grande place dans un

processus d'analyse et se doivent d'être modulaires afin de pouvoir se combiner les uns aux autres. Nos travaux sont organisés autour des 3 contributions principales suivantes :

1. **La rétro-ingénierie et ses problématiques** : Nous collectons et partageons les problématiques réelles des industriels lors de l'analyse de programmes binaires. Nous discutons des différents apports potentiels en termes d'automatisation et d'outillage pour aider les analystes à résoudre certaines de ces problématiques.

2. **Formalisation de Triton** : Le développement de Triton a été initié un an avant le début de la thèse mais cette dernière apporte une contribution de formalisation des composants internes ainsi que des analyses mises en place dans ce framework.

3. **Dévirtualisation binaire** : Nous avons publié [82, 83] une approche complètement automatique permettant la dévirtualisation de code binaire en combinant les fonctionnalités de Triton. De plus, nous discutons des limitations et des garanties de notre approche puis nous démontrons le potentiel de la méthode sur deux expérimentations. Nous mettons également en place un banc d'expérimentation permettant d'évaluer notre approche sur plusieurs formes de combinaisons de protection.

Durant ces 4 années de thèse, j'ai été amené à donner environ 140 heures de cours au sein de la formation BADGE à l'école ESIEA. Les thématiques abordées sont la recherche et l'exploitation de vulnérabilités, l'exécution symbolique, l'analyse de teinte et l'usage de Triton pour la rétro-ingéniérie. J'ai également donné des formations de 2 à 4 jours dans les locaux de Quarkslab sur l'usage de Triton. Ces formations ont été données pour initier le personnel au framework ainsi que pour des clients intéressés par le projet. Ces cours et ces formations m'ont permis de prendre du recul sur l'implémentation de nos analyses, d'avoir un retour d'expérience auprès de personnes travaillant sur les mêmes sujets mais aussi d'introduire Triton dans le monde de l'industrie.

Travaux futurs : Premièrement, les principaux projets sur lesquels mes efforts vont se concentrer sont le rajout d'un nouveau modèle mémoire afin de rendre la symbolisation de Triton complète ainsi que de modifier la façon d'appliquer la concrétisation et ainsi être correct. Ce nouveau modèle mémoire pourra être activé via un paramétrage afin de garder les deux modèles et de laisser le choix à l'analyste d'utiliser celui qui correspond au mieux à ses objectifs.

Deuxièmement, je vais travailler au sein de Quarkslab sur le développement d'une infrastructure de fuzzing pour la recherche de vulnérabilités. L'un des travaux futurs sera donc d'intégrer Triton dans le processus de recherche de vulnérabilités afin de créer un moteur d'exploration de chemins hybride (voir Section 3.2.3.5). La mise en place d'un moteur d'exploration hybride soulève plusieurs questions intéressantes sur lesquelles j'aimerais travailler telles que : *Comment définir le fait que l'exploration de chemins, à un moment X, n'est plus pertinente ? Comment définir si un chemin est plus pertinent à explorer qu'un autre ?*

Bibliographie

[1] Codevirtualizer : Total obfuscation against reverse engineering. `https://oreans.com/codevirtualizer.php`.

[2] Themida : Advanced windows software protection system. `https://www.oreans.com/themida.php`.

[3] Tigress : C diversifier/obfuscator. `http://tigress.cs.arizona.edu/`.

[4] Vmprotect : Software protection. `http://vmprotect.ru`.

[5] *Triton : A Dynamic Symbolic Execution Framework*. SSTIC, 2015.

[6] *Sibyl : Function Divination*. SSTIC, 2017.

[7] V. 35. Binary ninja : A new kind of reversing platform. `https://binary.ninja`, 2016–2018.

[8] V. M. Alvarez. Yara : The pattern matching swiss knife. 2008.

[9] B. Anckaert, M. Jakubowski, and R. Venkatesan. Proteus : virtualization for diversified tamper-resistance. In *Proceedings of the ACM workshop on Digital rights management*, pages 47–58. ACM, 2006.

[10] Avast. RetDec : An open-source machine-code decompiler. Botconf 2017, RECon 2018.

[11] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. Aeg : Automatic exploit generation. 2011.

[12] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *International conference on compiler construction*, pages 5–23. Springer, 2004.

[13] G. Balakrishnan and T. Reps. Wysinwyx : What you see is not what you execute. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(6) :23, 2010.

[14] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3) :50, 2018.

[15] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 189–200. ACM, 2016.

[16] S. Bardin, R. David, and J.-Y. Marion. Backward-bounded dse : Targeting infeasibility questions on obfuscated codes. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 633–651. IEEE, 2017.

[17] S. Bardin and P. Herrmann. Pruning the search space in path-based test generation. In *2009 International Conference on Software Testing Verification and Validation*, pages 240–249. IEEE, 2009.

[18] S. Bardin and P. Herrmann. OSMOSE : automatic structural testing of executables. *Softw. Test., Verif. Reliab.*, 21(1) :29–54, 2011.

[19] S. Bardin, P. Herrmann, and F. Védrine. Refinement-based cfg reconstruction from unstructured programs. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 54–69. Springer, 2011.

[20] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. Cvc4. In *International Conference on Computer Aided Verification*, pages 171–177. Springer, 2011.

[21] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[22] P. Biondi, R. Rigo, S. Zennou, and X. Mehrenberger. Bincat : purrfecting binary static analysis. In *Symposium sur la sécurité des technologies de l'information et des communications*, 2017.

[23] P. Boonstoppel, C. Cadar, and D. Engler. Rwset : Attacking path explosion in constraint-based test generation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 351–366. Springer, 2008.

[24] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 265–275. IEEE, 2003.

[25] R. Brummayer and A. Biere. Boolector : An efficient smt solver for bit-vectors and arrays. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–177, 2009.

[26] C. Cadar, D. Dunbar, and D. R. Engler. KLEE : unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224, 2008.

[27] C. Cadar, D. Dunbar, D. R. Engler, et al. Klee : Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[28] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE : automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, Ioctober 30 - November 3, 2006*, pages 322–335, 2006.

[29] C. Cadar and K. Sen. Symbolic execution for software testing : three decades later. *Commun. ACM*, 56(2) :82–90, 2013.

[30] J. Clause, W. Li, and A. Orso. Dytan : a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 196–206. ACM, 2007.

[31] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–196. ACM, 1998.

[32] K. Coogan, G. Lu, and S. Debray. Deobfuscation of virtualization-obfuscated software : a semantics-based approach. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 275–284. ACM, 2011.

[33] R. David. *Formal Approaches for Automatic Deobfuscation and Reverse-engineering of Protected Codes*. PhD thesis, Université de Lorraine, 2017.

[34] R. David, S. Bardin, J. Feist, L. Mounier, M.-L. Potet, T. D. Ta, and J. Marion. Specification of concretization and symbolization policies in symbolic execution. In *Proceedings of the 2016 International Symposium on Software Testing and Analysis*, ISSTA 2016. ACM, 2016.

[35] R. David, S. Bardin, T. D. Ta, L. Mounier, J. Feist, M.-L. Potet, and J.-Y. Marion. Binsec/se : A dynamic symbolic execution toolkit for binary-level analysis. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 653–656. IEEE, 2016.

[36] L. De Moura and N. Bjørner. Z3 : An efficient smt solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[37] F. Desclaux. Miasm : Framework de reverse engineering. *Actes du SSTIC. SSTIC*, 2012.

[38] A. Djoudi and S. Bardin. Binsec : Binary code analysis with low-level regions. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 212–217. Springer, 2015.

[39] A. Djoudi, S. Bardin, and É. Goubault. Recovering high-level conditions from binary programs. In *International Symposium on Formal Methods*, pages 235–253. Springer, 2016.

[40] B. Dutertre and L. De Moura. The yices smt solver. *Tool paper at http ://yices. csl. sri. com/tool-paper. pdf*, 2(2) :1–2, 2006.

[41] N. Eyrolles. *Obfuscation par expressions mixtes arithmético-booléennes : reconstruction, analyse et outils de simplification.* PhD thesis, Paris Saclay, 2017.

[42] N. Eyrolles, A. Guinet, and M. Videau. Arybo : Manipulation, canonicalization and identification of mixed boolean-arithmetic symbolic expressions. GreHack, France, Grenoble, 2016.

[43] B. Farinier, R. David, S. Bardin, and M. Lemerre. Arrays made simpler : An efficient, scalable and thorough preprocessing. *EPiC Series in Computing*, 57 :363–380, 2018.

[44] A. Fromherz, K. S. Luckow, and C. S. Pasareanu. Symbolic arrays in symbolic pathfinder. *ACM SIGSOFT Software Engineering Notes*, 41(6) :1–5, 2016.

[45] S. Ghosh, J. D. Hiser, and J. W. Davidson. A secure and robust approach to software tamper resistance. In *International Workshop on Information Hiding*, pages 33–47. Springer, 2010.

[46] P. Godefroid. Compositional dynamic test generation. In *ACM Sigplan Notices*, volume 42, pages 47–54. ACM, 2007.

[47] P. Godefroid. Higher-order test generation. In *ACM SIGPLAN Notices*, volume 46, pages 258–269. ACM, 2011.

[48] P. Godefroid, J. de Halleux, A. V. Nori, S. K. Rajamani, W. Schulte, N. Tillmann, and M. Y. Levin. Automating software testing using program analysis. *IEEE Software*, 25(5) :30–37, 2008.

[49] P. Godefroid, N. Klarlund, and K. Sen. DART : directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 213–223, 2005.

[50] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*, 2008.

[51] P. Godefroid, M. Y. Levin, and D. A. Molnar. SAGE : whitebox fuzzing for security testing. *Commun. ACM*, 55(3) :40–44, 2012.

[52] Heaventools. Pe-explorer : View, edit, and reverse engineer exe and dll files. `http://www.heaventools.com`, 2000.

[53] C. Heffner. Binwalk : Firmware analysis tool. 2010.

[54] Hex-Rays. IDA : A multi-processor disassembler and debugger. `https://www.hex-rays.com/products/ida/index.shtml`, 2005–2017.

[55] S.-K. Huang, M.-H. Huang, P.-Y. Huang, C.-W. Lai, H.-L. Lu, and W.-M. Leong. Crax : Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations. In *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*, pages 78–87. IEEE, 2012.

[56] K. Jamrozik, G. Fraser, N. Tillman, and J. De Halleux. Generating test suites with augmented dynamic symbolic execution. In *International Conference on Tests and Proofs*, pages 152–167. Springer, 2013.

[57] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 215–224, 2010.

[58] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin. Obfuscator-LLVM – software protection for the masses. In B. Wyseur, editor, *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, pages 3–9. IEEE, 2015.

[59] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. Dta++ : dynamic taint analysis with targeted control-flow propagation. In *NDSS*, 2011.

[60] J. Kinder. Towards static analysis of virtualization-obfuscated binaries. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 61–70. IEEE, 2012.

[61] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7) :385–394, 1976.

[62] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *Acm Sigplan Notices*, volume 47, pages 193–204. ACM, 2012.

[63] C. Lattner and V. Adve. LLVM : A compilation framework for lifelong program analysis and transformation. In *CGO*, pages 75–88, San Jose, CA, USA, Mar 2004.

[64] P. Lestringant. *Identification d'algorithmes cryptographiques dans du code natif*. PhD thesis, Rennes 1, 2017.

[65] Y. Liu, X. Zhou, and W.-W. Gong. A survey of search strategies in the dynamic symbolic execution. In *ITM Web of Conferences*, volume 12, page 03025. EDP Sciences, 2017.

[66] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks. Directed symbolic execution. In *International Static Analysis Symposium*, pages 95–111. Springer, 2011.

[67] R. Majumdar and K. Sen. Hybrid concolic testing. In *29th International Conference on Software Engineering (ICSE'07)*, pages 416–426. IEEE, 2007.

[68] Maximus. Reversing a simple virtual machine. CodeBreakers 1.2, 2006.

[69] X. Meng and B. P. Miller. Binary code is not easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 24–35. ACM, 2016.

[70] J. Ming, D. Xu, L. Wang, and D. Wu. Loop : Logic-oriented opaque predicate detection in obfuscated binary code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 757–768. ACM, 2015.

[71] J. Nagra and C. Collberg. *Surreptitious software : obfuscation, watermarking, and tamperproofing for software protection.* Pearson Education, 2009.

[72] N. Nethercote and J. Seward. Valgrind : a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[73] J. Palsberg, S. Krishnaswamy, M. Kwon, D. Ma, Q. Shao, and Y. Zhang. Experience with software watermarking. In *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)*, pages 308–316. IEEE, 2000.

[74] Pancake and al. Radare : Unix-like reverse engineering framework and commandline tools. `http://www.radare.org`, 2011–2018.

[75] D. M. Perry, A. Mattavelli, X. Zhang, and C. Cadar. Accelerating array constraints in symbolic execution. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 68–78. ACM, 2017.

[76] C. S. Păsăreanu, N. Rungta, and W. Visser. Symbolic execution with mixed concrete-symbolic solving. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 34–44, New York, NY, USA, 2011. ACM.

[77] N. A. Quynh. Capstone : Next-gen disassembly framework. *Black Hat USA*, 2014.

[78] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn. Pin : a binary instrumentation tool for computer architecture research and education. In *Proceedings of the 2004 workshop on Computer architecture education : held in conjunction with the 31st International Symposium on Computer Architecture*, page 22. ACM, 2004.

[79] J. Robbins. Debugging windows based applications using windbg. *Miscrosoft Systems Journal*, 1999.

[80] R. Rolles. Unpacking virtualization obfuscators. In *3rd USENIX Workshop on Offensive Technologies.(WOOT)*, 2009.

[81] M. Rosenblum. Vmwares virtual platform. In *Proceedings of hot chips*, volume 1999, pages 185–196, 1999.

[82] J. Salwan, S. Bardin, and M.-L. Potet. Deobfuscation of vm based software protection. In *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, June 7-9 2017*, pages 119–142. SSTIC, 2017.

[83] J. Salwan, S. Bardin, and M.-L. Potet. Symbolic deobfuscation : From virtualized code back to the original. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 372–392. Springer, 2018.

[84] Scherzo. Inside code virtualizer. `https://tuts4you.com/download.php?view.2640`, 2007.

[85] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl. Protecting software through obfuscation : Can it keep pace with progress in code analysis ? *ACM Computing Surveys (CSUR)*, 49(1) :4, 2016.

[86] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 317–331, 2010.

[87] K. Sen, D. Marinov, and G. Agha. CUTE : a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 263–272, 2005.

[88] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee. Automatic reverse engineering of malware emulators. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, pages 94–109, 2009.

[89] R. Stallman and al. GDB : The gnu project debugger. `https://sourceware.org/gdb/`, 1986–2018.

[90] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller : Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[91] R. Thomas. Lief - library to instrument executable formats. `https://github.com/lief-project/LIEF`, 2016.

[92] N. Tillmann and J. de Halleux. Pex-white box test generation for .net. In *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, pages 134–153, 2008.

[93] R. Tofighi-Shirazi, M. Christofi, P. Elbaz-Vincent, and T.-H. Le. Dose : Deobfuscation based on semantic equivalence. In *Proceedings of the 8th Software Security, Protection, and Reverse Engineering Workshop*, page 1. ACM, 2018.

[94] S. K. Udupa, S. K. Debray, and M. Madou. Deobfuscation : Reverse engineering obfuscated code. In *12th Working Conference on Reverse Engineering (WCRE'05)*, pages 10–pp. IEEE, 2005.

[95] A. Velte and T. Velte. *Microsoft Virtualization with Hyper-V*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2010.

[96] N. Williams, B. Marre, and P. Mouy. On-the-fly generation of k-path tests for C functions. In *19th IEEE International Conference on Automated Software Engineering ASE, 20-25 September 2004, Linz, Austria*, pages 290–293, 2004.

[97] B. Yadegari and S. Debray. Bit-level taint analysis. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 255–264. IEEE, 2014.

[98] B. Yadegari and S. Debray. Symbolic execution of obfuscated code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 732–744, 2015.

[99] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. A generic approach to automatic deobfuscation of executable code. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 674–691. IEEE, 2015.

[100] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim. {QSYM} : A practical concolic execution engine tailored for hybrid fuzzing. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 745–761, 2018.

[101] Y. Zhou and A. Main. Diversity via code transformations : A solution for ngna renewable security. *NCTA-The National Show*, 2006.

[102] Y. Zhou, A. Main, Y. X. Gu, and H. Johnson. Information hiding in software with mixed boolean-arithmetic transforms. In *International Workshop on Information Security Applications*, pages 61–75. Springer, 2007.

[103] zyantific. Zydis : The ultimate, open-source x86 and x86-64 decoder/disassembler library. *Black Hat USA*, 2018.