UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Analyse du flot de données pour la construction du graphe de flot de contrôle des codes obfusqués

THÈSE

présentée et soutenue publiquement le 22 février 2021

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Sylvain Cecchetto

Composition du jury

Rapporteurs :	Valérie Viet Triem Tong	CentraleSupélec - Rennes
	José Fernandez	Polytechnique Montréal
Examinateurs :	Nadia Tawbi	Université Laval Québec
	Sarah Zennou	Airbus
	Christophe Hauser	Université de Caroline du Sud
	Stephan Merz	Université de Lorraine
Invités :	Sébastien Bardin	CEA LIST
	Colas Le Guernic	Verimag
Encadrants :	Jean-Yves Marion	Université de Lorraine
	Guillaume Bonfante	Université de Lorraine

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Thèse soutenue financièrement par la Direction générale de l'armement

DGA

Loria Laboratoire lorrain de recherche en informatique et ses applications

Remerciements

Je tiens tout d'abord à remercier mes directeurs de thèse : Jean-Yves Marion et Guillaume Bonfante. Merci pour la confiance que vous m'avez accordée durant ces années et merci également d'avoir partagé vos connaissances avec moi sur ce sujet passionnant. Fabrice, merci également à toi pour ton aide et tes conseils en ce qui concerne la « technique ».

Je souhaite également remercier les membres du jury pour l'intérêt qu'ils ont porté à mon travail. Merci pour le temps que vous avez consacré à l'évaluation de ma thèse mais aussi pour votre disponibilité malgré les conditions particulières de la soutenance.

Stephan Merz, merci d'avoir accepté d'endosser le rôle de président du jury et merci également de m'avoir accompagné durant cette thèse à travers votre rôle de référent scientifique.

Si ce mémoire voit le jour aujourd'hui c'est aussi et beaucoup grâce à toi Caroline. Tu m'as poussé à démarrer cette thèse et surtout tu as été d'un soutien infaillible durant toutes ces années malgré les hauts et les bas. Un énorme merci à toi !

Bien sûr je souhaite également remercier mes parents, depuis toujours vous m'accordez une confiance aveugle et j'ai eu la chance de pouvoir réaliser les études que je voulais et où je voulais sans jamais avoir à me poser de questions. Merci pour votre soutien qui a toujours été là malgré la distance.

Plus généralement je souhaite remercier mes grands parents ainsi que toute ma famille. Mais aussi Gisèle, Michel et toute la famille de Caroline. Merci à vous tous pour votre soutien.

Merci également à toutes les personnes qui participent à la vie et au fonctionnement du Loria car il faut le reconnaître, j'ai eu la chance de travailler dans un environnement exceptionnel. À cela s'ajoute les multiples pauses café avec Quentin, Jean-Philippe, Kevin et les autres stagiaires et doctorants des différentes équipes. De vrais moments de pauses mais aussi d'entraide et de travail. Or du laboratoire, les différentes formations doctorales ont été d'autant plus intéressantes qu'elles m'ont permis de faire la rencontre de nouvelles amies.

Merci également à mes amis du Lycée : Julia, Massouille, Jérôme et Kdo, malgré la distance vous répondez toujours présent lors de nos passages dans le sud.

Enfin je souhaite remercier Régis, Jean-Yves, Guillaume et Fabrice car vous m'offrez la possibilité de continuer à travailler dans le domaine des logiciels malveillants.

Sommaire

Table des figures

Liste des codes

Liste des tableaux

Chapitre 1

Introduction

1.1 Contexte

1.1.1 Histoire de virus

En 1986, dans leur boutique informatique, deux frères pakistanais créent le virus *Brain*. Ayant remarqué que leurs clients redistribuaient leurs logiciels, les vendeurs avaient décidé de créer ce virus afin de leur donner une leçon. *Brain* parviendra à infecter des ordinateurs du monde entier alors même qu'il utilise comme moyen de propagation une simple disquette [16]. Plus tard, la démocratisation de l'accès à internet offrira aux virus un moyen de propagation de choix. On se souviendra notamment du virus *I Love You*, qui, propagé massivement par mail occasionnera des dégâts d'un coût estimé à environ 8 milliards de dollars [48]. Au fil du temps, le panorama des menaces évolue à la hausse et les virus cohabitent maintenant avec des chevaux de Troie, des rançongiciels et autres vers en tout genre, on parle dès lors de logiciels malveillants ou *malwares*. De nos jours, ces menaces ont souvent un but lucratif, cela a par exemple été le cas lors de la médiatisée cyberattaque mondiale opérée par *WannaCry*. En 2017, ce rançongiciel demandera aux utilisateurs de 300 000 ordinateurs dans plus de 150 pays un paiement en échange de la restitution de leurs fichiers chiffrés. Il mettra également à l'arrêt de nombreuses organisations et entreprises comme Renault en France [75]. En 2015, la chaîne internationale TV5 Monde est victime d'une attaque ciblée et savamment préparée entraînant notamment l'arrêt de sa diffusion pendant plusieurs heures. Revendiquée par un groupe se réclamant de l'État islamique, les enquêteurs s'orientent finalement sur la piste du groupe de pirates *APT28*, soupçonné d'être le bras armé du Kremlin sur internet [46]. Parmi tant d'autres, cette affaire nous prouve que les menaces informatiques d'aujourd'hui sont le fruit d'une véritable industrie du *malware* dont la dimension économique et politique est bien présente.

1.1.2 Antivirus

Dès le début des années 90, les premiers logiciels antivirus font leur apparition sur le marché dans le but de nous prémunir des logiciels malveillants. Leurs objectifs annoncés sont simples : détecter, neutraliser et enfin éliminer les éventuelles menaces dont peut être victime un système informatique. Il faut cependant noter que leur tâche n'est pas aisée, en effet le code source des logiciels malveillants n'est en général pas disponible, les antivirus doivent alors se contenter des fichiers exécutables pour leurs analyses. La présence de ces outils a conduit les éditeurs de *malwares* et les experts en sécurité informatique à entrer dans un jeu perpétuel du chat et de la souris. Alors que les méthodes de détection des antivirus sont de plus en plus efficaces, l'industrie du *malware* met en place différentes techniques de protections et de contre-mesures afin d'échapper aux antivirus et ainsi rester indétectable. Parmi elles, l'obfuscation regroupe l'ensemble des méthodes visant à rendre difficile et à ralentir le processus de rétro-ingénierie des fichiers binaires. Cette technique s'attaque aussi bien aux méthodes d'analyses automatiques qu'au travail manuel d'un informaticien spécialiste. Initialement utilisée pour protéger le code des logiciels propriétaires comme les jeux vidéos, l'obfuscation est maintenant une des techniques largement répandue dans le domaine des logiciels malveillants. La nécessité de détecter, d'atténuer et d'outre passer ce genre de contre-mesures

est devenue une des priorités dans le domaine de l'analyse de logiciels malveillants.

1.1.3 Analyse des logiciels malveillants

Il existe deux grands types de méthodes pour l'analyse des logiciels malveillants : l'analyse dite statique et l'analyse dynamique. La différence principale entre ces deux types d'analyse réside dans le fait que le binaire étudié sera exécuté lors d'une analyse dynamique, ce qui n'est pas le cas dans le cadre d'une analyse statique.

Les postes informatiques terminaux sont généralement équipés de solutions antivirus reposant sur une analyse statique. Habituellement, ces antivirus commerciaux utilisent une détection basée sur la recherche, dans le binaire analysé, d'une portion de code connue comme appartenant à un logiciel malveillant. On parle ici d'une détection par recherche de signature virale [13]. D'un autre côté, le domaine de l'apprentissage automatique n'a cessé d'évoluer ces dernières années. Par conséquent, de nombreux travaux sur l'analyse de *malwares* comme la détection ou la classification utilisent cette technologie [70, 33].

Concernant les analyses dynamiques, différentes approches existent. On pourra par exemple surveiller le binaire au plus près de ses actions en utilisant des techniques d'instrumentation [23] ou alors prendre un peu plus de recul en observant ses interactions avec son environnement [39] et le monde extérieur [3].

L'exécution symbolique est une approche d'analyse statique de plus en plus présente dans le domaine du logiciel malveillant. Cette méthode formelle permet de calculer une formule contenant un ensemble de relations pour un chemin donné dans un programme. Il est possible de cette façon d'explorer l'ensemble des chemins afin de couvrir la totalité du programme. On obtient alors les différents jeux d'entrée permettant l'exécution des différents chemins [44, 4].

Il est possible d'utiliser une exécution symbolique en tandem avec une analyse dynamique, on effectue alors ce que l'on appelle une exécution concolique [34, 65, 74]. Les avantages de cette technique sont nombreux comme la résolution des sauts indirects. Néanmoins la couverture de l'analyse sera sous-approximée car dépendante des entrées utilisées lors de l'analyse dynamique.

1.1.4 Enjeux

Le travail de cette thèse porte sur le développement d'une nouvelle méthode d'analyse statique des logiciels malveillants, ou plus largement sur l'analyse des fichiers exécutables en général. Nous avons choisi de baser notre approche sur une analyse statique et symbolique pour plusieurs raisons :

- **Limites des approches dynamiques :** Les analyses dynamiques sont généralement assez robustes face à certaines obfuscations comme l'auto-modification. Cependant les éditeurs de *malwares* réussissent de mieux en mieux à protéger leurs binaires contre ce genre d'analyses en y intégrant différentes heuristiques de détection pour changer le comportement de leur logiciel malveillant si ce dernier détecte la présence d'un environnement d'analyse.
- **Facilité de mise en œuvre :** Une analyse de type statique est généralement plus facile à mettre en œuvre qu'une analyse dynamique. En effet, étant donné qu'une analyse dynamique conduit à exécuter le fichier à analyser, il est en général nécessaire de mettre en place un environnement contrôlé (appelé *sandbox*) comme une machine virtuelle afin d'éviter les risques comme la propagation du potentiel *malware*.
- **Transparence de l'analyse :** Certains logiciels malveillants nécessitent un accès à internet durant leur exécution. Si l'environnement d'exécution d'une analyse dynamique bloque l'accès à internet, il est probable que le binaire analysé avorte son exécution rendant ainsi l'analyse incomplète. D'un autre coté, si l'on ouvre l'accès à internet alors l'attaquant peut savoir que son *malware* est actuellement exécuté. Ce genre d'information peut s'avérer utile dans le cadre d'une attaque ciblée. Ce risque n'existe pas lors d'une exécution symbolique car le binaire n'est jamais exécuté réellement.
- **Couverture d'analyse :** L'exécution symbolique permet l'exploration d'une plus grande couverture du graphe de flot de contrôle du binaire car différents chemins d'exécution peuvent être explorés durant l'analyse. Au contraire, une analyse dynamique va généralement effectuer une exécution unique du binaire représentant ainsi qu'un seul chemin d'exécution.

1.2 Contributions

1.2.1 Plateforme BOA

Cette thèse a permis le développement de la platefrome BOA, pour *Basic blOck Analysis*, un outil dont le but est de construire le graphe de flot de contrôle d'un binaire quelconque. Notons cependant que BOA n'a pas vocation à remplacer un antivirus, il ne fait pas de détection. Il faut plutôt le voir comme un allié aux antivirus que l'on placerait en amont de ces derniers. La figure 1.1 montre un exemple d'utilisation de BOA au sein d'une plateforme d'analyse antivirus. Dans cette plateforme fictive, l'analyse de BOA est utilisée en tandem avec une analyse dynamique ainsi qu'un désassemblage statique classique. Un module d'agrégation récupère ensuite les résultats de ces trois outils comme la liste des instructions désassemblées, les différentes obfuscations détectées ou encore la liste des entrées de la table d'importation. L'agrégateur fusionne ces résultats avant de les soumettre à un module de détection antivirus qui va permettre de conclure si le fichier exécutable donné en entrée de la plateforme est considéré comme malveillant ou non.

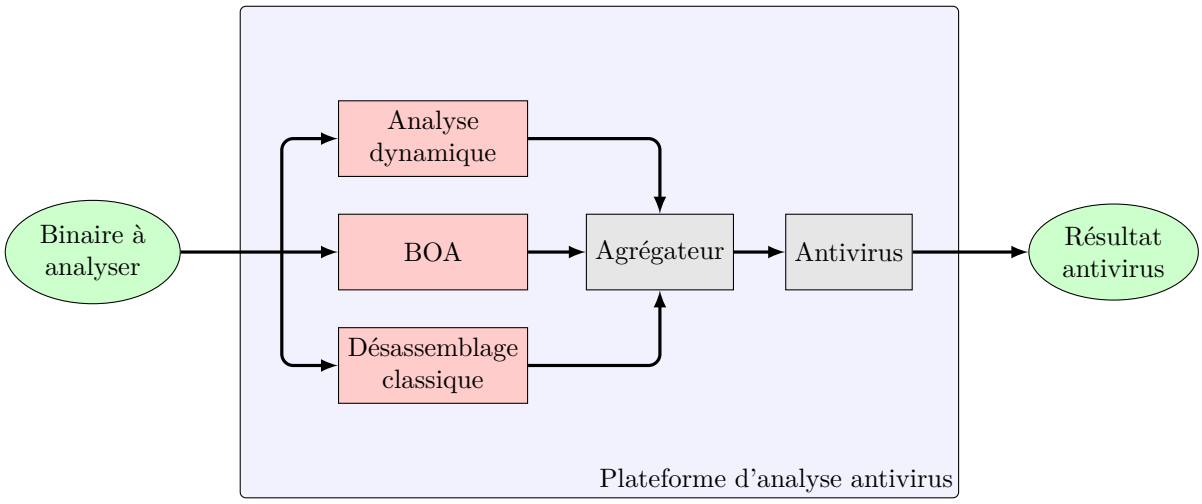


FIGURE 1.1 – Exemple d'utilisation de BOA dans une plateforme d'analyse antivirus

Nous avons fait le choix de s'orienter vers une approche entièrement statique et symbolique. Nous souhaitons montrer que ce duo est tout à fait capable de se mesurer à des binaires obfusqués avec des résultats supérieurs aux analyses statiques traditionnelles et convaincants par rapport aux résultats que l'on obtient à la suite d'une analyse dynamique.

Concrètement, BOA est basé sur un fonctionnement itératif où chaque tour va permettre le désassemblage (si besoin) et l'exécution symbolique d'un bloc de base. À l'issue de cette exécution symbolique, un état partiel de la machine est calculé, nous permettant ainsi de connaître l'adresse successeur de ce bloc de base conduisant ainsi au tour suivant de la boucle.

Le schéma fonctionnel simplifié est donné en figure 1.2. BOA représente environ 10 400 lignes de code C++ et le fonctionnement général de BOA est expliqué dans le chapitre 5.

L'ensemble des contributions que nous apportons sont implémentées dans BOA. Nous avons notamment défini une sémantique opérationnelle à continuation supportant les codes auto-modifiants et permettant également la simulation du système d'exploitation pour la gestion des fonctions externes et des exceptions. Nous avons également défini un système à piles afin de détecter les boucles dans un graphe de flot de contrôle mais aussi pour la détection des falsifications de la pile d'appels.

1.2.2 Sémantique

La première contribution de cette thèse concerne la définition d'une sémantique. Pour les besoins du fonctionnement de BOA, notre sémantique doit être capable de gérer les codes auto-modifiants mais

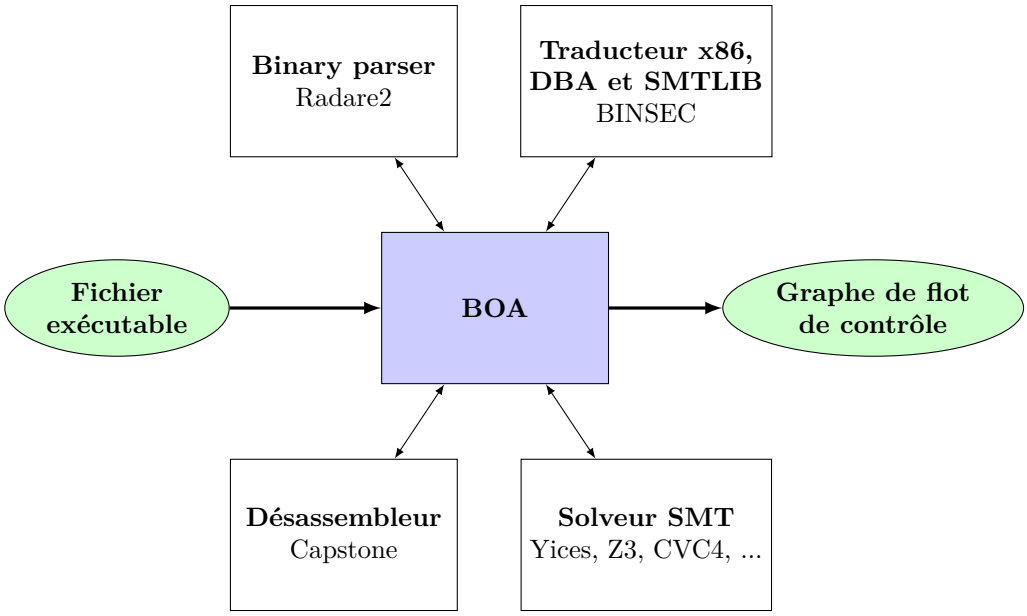


FIGURE 1.2 – Schéma fonctionnel de la plateforme BOA

aussi être en mesure de simuler les interactions entre le programme et le système d'exploitation. Cette sémantique est définie dans le chapitre 3.

1.2.2.1 Gestion des codes auto-modifiants

Durant l'analyse d'un programme, il est courant de voir ce dernier comme un élément fixe et externe à la machine [40]. Si cette vision des choses convient dans le cadre d'une étude sur des binaires bénins, elle n'est généralement pas adaptée aux *malwares*. En effet, les logiciels malveillants utilisent souvent des techniques d'auto-modification conduisant le programme à se modifier lui-même à la volée au cours de son exécution, comportement qui n'est traditionnellement pas supporté par une sémantique opérationnelle classique.

Pour cette raison, nous avons fait le choix dans notre sémantique de considérer un programme comme une simple suite d'octets de données. En utilisant cette approche, le programme n'est plus un élément fixe et externe à la machine mais il fait parti de la machine en tant que donnée. Concrètement, afin d'interpréter un programme avec cette approche, on construit l'état initial de la machine en « chargeant » le programme dans la machine. Ensuite, le programme en tant que tel (un fichier PE par exemple) n'est plus utilisé, il vit maintenant au sein de la machine. La suppression de cette frontière entre le programme et les éléments de la machine permet par définition de supporter les codes auto-modifiants ; en effet, comme le programme vit dans la machine, il a la possibilité de se modifier au cours de son exécution.

1.2.2.2 Sémantique à continuation

En sémantique des langages de programmation, la sémantique opérationnelle est régulièrement utilisée. Généralement ce genre d'approche permet de décrire comment l'exécution d'un programme se traduit en termes de suite d'états successifs de la machine. Cependant, le système d'exploitation n'est généralement pas pris en compte.

Pour nous, un programme ne peut vivre sans ses relations avec le système d'exploitation. Pour cette raison, nous avons fait le choix de définir dans BOA une sémantique opérationnelle à continuation [32, 62]. La continuation va être chargée de calculer l'état futur de la machine à partir d'un état machine courant. Elle permet concrètement de traiter les actions qui dépendent du système d'exploitation et qui sont en dehors de la stricte sémantique du processeur. Cette sémantique nous permet de simuler deux

fonctionnalités normalement gérées par le système d'exploitation : la détection et la gestion des exceptions ainsi que les appels systèmes.

Gestion des exceptions L'utilisation du mécanisme des exceptions est courante dans les programmes. Pour cette raison, nous avons souhaité que BOA soit en mesure de gérer cette fonctionnalité. Pour cela, nous avons défini notre sémantique à continuation de telle façon qu'elle soit en mesure de (i) détecter une exception déclenchée par le fichier exécutable analysé et (ii) simuler le comportement du système d'exploitation, à savoir la recherche dans le binaire d'un gestionnaire d'exception pouvant gérer l'exception en cours ainsi que la sauvegarde du contexte d'exécution sur la pile. Le schéma 1.3 montre les actions effectuées par la sémantique à continuation dans le cadre de la gestion des exceptions Windows.

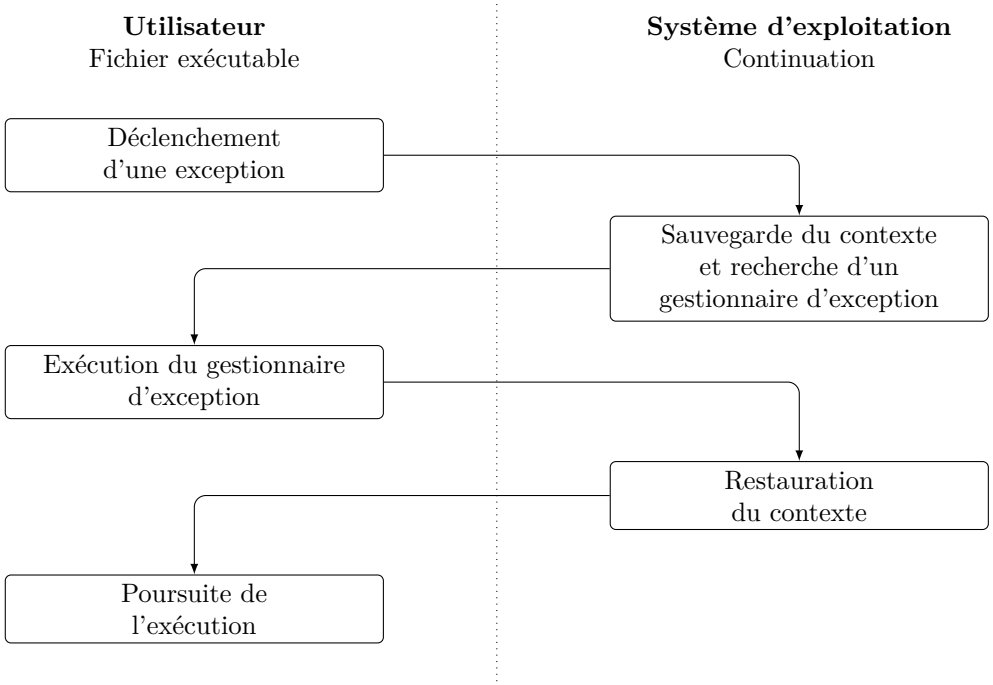


FIGURE 1.3 – Gestion des exceptions Windows 32 bits

Afin de rendre la détection des exceptions possible, nous avons notamment dû définir la notion de permissions des cases mémoire de notre représentation machine. Cette propriété supplémentaire nous permet, au cours de l'exécution symbolique d'une instruction, de vérifier que son code opérationnel vit sur des cases mémoire dont la permission d'exécution est bien présente. De la même façon, nous pouvons surveiller que les opérations de lecture et d'écriture en mémoire sont effectuées sur des cases dont la permission le permet.

La gestion des exceptions effectuée par BOA est donnée dans le chapitre 10.

Simulation des fonctions externes Il est courant que les logiciels, durant leur exécution, fassent appel à des fonctions externes, c'est par exemple le cas lorsqu'un programme utilise une fonction de l'API Windows. Lors d'une compilation dynamique, le code de ces fonctions externes n'est pas embarqué dans le programme et c'est seulement durant l'exécution que le système d'exploitation va récupérer le code des fonctions externes nécessaire au programme et faire le lien entre les deux.

Pour gérer les appels aux fonctions externes, BOA embarque un système de *hook* afin de détecter l'appel à une fonction externe ainsi qu'un système de *stub* permettant de simuler l'impact de l'exécution de cette fonction sur l'état de la machine en fonction de la valeur des arguments avec laquelle elle est appelée. La simulation des fonctions les plus courantes de l'API Windows est déjà implémentée dans BOA mais n'importe quel utilisateur peut facilement modifier ou ajouter ses propres *hook* et *stub* qui

seront déclenchés en fonction du nom de la fonction externe et de la bibliothèque dans laquelle elle est normalement présente.

Le fonctionnement de cette simulation est expliqué dans le chapitre 9.

1.2.3 Système à pile

Dans la section 3.4 nous définissons un système à pile dont les états correspondent aux blocs de base d'un graphe de flot de contrôle auquel il est lié. Grâce aux règles de transition que nous avons définies, ce système à pile nous permet de définir une notion de boucle. Ainsi, durant l'analyse de BOA, nous pouvons détecter les boucles dans le graphe de flot de contrôle mais aussi détecter si des falsifications de la pile d'appels ont lieu.

1.2.4 Exécution symbolique et analyse de teinte

1.2.4.1 Langage intermédiaire et formule SMT

Afin d'effectuer l'exécution symbolique des instructions x86, nous avons fait le choix d'utiliser le langage intermédiaire DBA [29] utilisé dans la plateforme BINSEC nous facilitant ainsi la traduction des instructions en formules SMT équivalentes. Le schéma 1.4 montre un exemple de cette traduction.

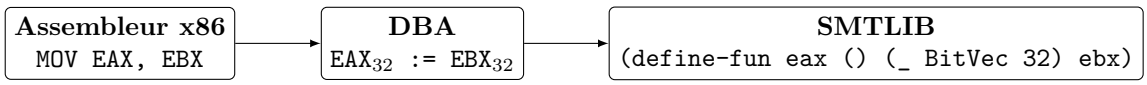


FIGURE 1.4 – Exemple d'une traduction assembleur x86, DBA et SMTLIB

Durant son analyse, BOA passe la majorité de son temps dans l'exécution symbolique à travers l'utilisation d'un solveur SMT et cette tâche est relativement coûteuse en temps. Pour ces raisons, nous avons optimisé ce module de BOA afin d'obtenir des temps de traitement les plus cours possibles notamment en utilisant un algorithme d'analyse de teinte.

Dans notre cas, nous utilisons une exécution symbolique afin de pouvoir calculer l'état de la machine (valeurs des registres et des cases mémoire) après l'exécution d'une ou plusieurs instructions à partir d'un état machine initial.

Nous avons fait le choix d'effectuer l'exécution symbolique à l'échelle du bloc de base. Nous pensons que cette granularité est un bon compromis entre une exécution à l'échelle de l'instruction qui sera certes rapide mais demandera un nombre d'appels au solveur SMT plus important, et une exécution à l'échelle de plusieurs blocs de base qui engendrerait un traitement long du solveur SMT.

1.2.4.2 Analyse de teinte

Toujours dans le but de limiter le temps de traitement du solveur SMT, nous avons également implémenté un algorithme d'analyse de teinte. Cet algorithme nous permet deux choses :

- Tout d'abord, cette analyse nous permet de calculer l'état machine minimal nécessaire à l'exécution symbolique d'un bloc de base. L'état machine minimal correspond aux seuls éléments (valeurs des registres et des cases mémoire) nécessaires aux instructions du bloc de base à exécuter. En effet, il est peu probable que durant son exécution un bloc de base effectue une lecture de la totalité des registres et des cases mémoire de la machine. L'utilisation d'un état machine minimal nous permet de réduire le nombre de données à fournir au solveur SMT et cela permet donc de faciliter son traitement.
- L'analyse de teinte nous permet également de connaître les éléments de la machine qui ne seront pas modifiés par l'exécution du bloc de base. Inutile alors pour ces registres et cases mémoire de solliciter le solveur SMT. Pour les éléments de la machine qui sont modifiés, l'analyse de teinte va nous permettre, en fonction de l'état machine d'entrée, de savoir si la valeur de ces éléments sera connue ou non après l'exécution. Nous savons ainsi qu'il ne sera pas nécessaire de demander au solveur SMT de calculer la valeur des registres et cases mémoire que nous savons inconnue, nous économisant encore des appels au solveur SMT.

1.2.5 Gestion des obfuscations et validations expérimentales

Les logiciels malveillants utilisent différentes techniques d'obfuscations, nous avons alors implémenté BOA de façon à détecter et supporter certaines d'entre-elles.

1.2.5.1 Sauts dynamiques et falsifications de la pile d'appels

En fonction d'un contexte initial donné, l'algorithme principal de BOA permet le calcul des adresses de saut des instructions de saut dynamique rencontrées durant l'exploration des différents chemins d'exécution. Les détails de cette fonctionnalité mais aussi la technique de détection des falsifications de la pile d'appels sont discutés dans le chapitre 6. Nous avons évalué ces fonctionnalités de BOA sur un jeu de binaires non protégés mais également sur des binaires obfusqués à l'aide de l'obfuscateur de code source Tigress ainsi que sur des binaires packés par des packers Windows commerciaux. La totalité de ces expériences a abouti à un succès.

1.2.5.2 Prédicats opaques et branches mortes

L'algorithme de BOA permet de vérifier que chaque branche d'un saut conditionnel est bien « vivante » avant de l'emprunter durant son analyse et de l'ajouter dans le graphe de flot de contrôle. Indépendamment à cette fonctionnalité, nous avons également ajouté dans BOA une analyse de détection des prédicats opaques qui fonctionne à l'échelle du bloc de base. Ces méthodes ont été évaluées sur un jeu de programmes protégés par Tigress ainsi que sur un échantillon du *malware* XTunnel connu pour contenir beaucoup de prédicats opaques. Notons que nous avons également utilisé une instance de Grid'5000 [5] afin de réaliser cette expérience. Ces résultats sont donnés dans le chapitre 7.

1.2.5.3 Codes auto-modifiants et gestion des vagues

La chapitre 8 décrit la façon dont nous détectons les auto-modifications mais également comment nous modélisons notre approche en utilisant le concept de vagues d'exécutions décrit par Bonfante et al. [15]. Concrètement, la construction du graphe de flot de contrôle de BOA est basée sur une notion de vague permettant d'assigner à chaque instruction exécutée la vague d'auto-modification à laquelle elle appartient. Pour évaluer notre approche, nous avons soumis BOA au dépackage d'un binaire précédemment packé par 35 packers Windows différents et nous avons comparé nos résultats aux différentes analyses dynamiques connues. Parmi ces 35 binaires packés, nous avons réussi à obtenir le graphe de flot de contrôle de chacune de leurs vagues pour 14 d'entre eux. De plus, nous avons analysé un échantillon du cheval de Troie Emotet connu pour embarquer différentes phases de dépackage. Cette expérience nous a permis de récupérer la seconde vague d'exécution de ce binaire qui est bien détectée comme un échantillon de Emotet par VirusTotal.

1.2.5.4 Chargement des DLL et construction de la table d'importation à la volée

En ce qui concerne les binaires Windows, BOA permet de simuler le chargement des DLL utilisées par un binaire, que ces bibliothèques soient chargées au lancement du programme via la table d'importation ou bien à la volée par l'intermédiaire des fonctions de l'API Windows comme *LoadLibrary*. Grâce à cette fonctionnalité, et en utilisant également un mécanisme de *hook* des fonctions externes, BOA est capable de détecter la construction à la volée de la table d'importation d'un binaire protégé et ce en surveillant la chaîne de caractères utilisée comme argument lors des différents appels à la fonction *GetProcAddress* de l'API Windows. Cette technique est décrite dans le chapitre 9. Afin d'expérimenter cette technique, nous avons packé un binaire à l'aide de 35 packers Windows différents. Parmi ces 35 versions protégées, nous avons réussi à retrouver la table d'importation du binaire original pour 15 d'entre eux.

1.2.5.5 Exceptions Windows

Afin de compliquer leur analyse, certains logiciels malveillants font un usage non conventionnel du mécanisme d'exception. Concrètement, durant leur exécution, ces programmes déclenchent volontairement une exception afin d'exécuter un gestionnaire d'exception qu'ils ont préalablement mis en place. Durant

son exécution, le gestionnaire d'exception a la possibilité de modifier comme bon lui semble le contexte d'exécution précédemment sauvegardé par le système d'exploitation sur la pile. Lors d'une analyse statique, cette technique peut engendrer la construction d'une graphe de flot de contrôle incomplet mais aussi incorrect.

Grâce à la sémantique à continuation que nous avons implémentée, BOA est capable de détecter le déclenchement de ces exceptions mais aussi de simuler fidèlement le comportement de Windows en gérant la chaîne SEH, sauvegardant sur la pile le contexte d'exécution en cas d'exception et également en restaurant ce contexte à la fin de l'exécution d'un gestionnaire d'exception.

Ces fonctionnalités nous permettent de gérer correctement les différentes techniques d'obfuscations à base d'exceptions que l'on peut rencontrer dans certains packers Windows comme nous le montrons dans le chapitre 10.

1.3 Organisation du document

La suite du document est organisé de la façon suivante :

— Dans le chapitre 2 nous mettons en évidence la problématique que nous avons traitée dans ces travaux de thèse. Nous rappelons les difficultés rencontrées lors de l'analyse d'un fichier exécutable, notamment durant les phases de désassemblage et de construction du graphe de flot de contrôle. Nous décrivons également le fonctionnement des différentes obfuscations que nous traitons dans ce document.

— Dans le chapitre 3 nous décrivons le modèle machine que nous utilisons au sein de la plateforme BOA. Nous définissons ensuite une sémantique à continuation permettant de traiter les codes auto-modifiants mais également capable de simuler le système d'exploitation pour la gestion des exceptions et des appels systèmes. Enfin, nous définissons un système à pile, qui, utilisé conjointement au graphe de flot de contrôle auquel il est lié, nous permet de détecter les boucles ainsi que les falsifications de la pile d'appels.

— Le chapitre 4 est consacré à l'implémentation du moteur d'exécution que nous utilisons afin d'exécuter symboliquement une suite d'instructions x86. Pour cela nous commençons par présenter le langage intermédiaire DBA que nous utilisons afin de traduire une instruction x86 en une formule SMTLIB équivalente.

— Le chapitre 5 décrit le fonctionnement de BOA dans le cas général. Ce chapitre explique techniquement les différentes étapes réalisées par BOA afin qu'il produise le graphe de flot de contrôle d'un fichier exécutable.

— Dans le chapitre 6 nous revenons sur la façon dont BOA calcule les adresses de saut des instructions de saut indirect et nous décrivons également la méthode que nous mettons en œuvre afin de détecter les falsifications de la pile d'appels.

— Le chapitre 7 est consacré à la détection et à la gestion des prédicats opaques présents dans les binaires obfusqués. Plus généralement nous expliquons comment BOA parvient a détecter les branches mortes d'un programme.

— La détection des codes auto-modifiants ainsi que la gestion sous forme de vague sont traités dans le chapitre 8.

— Dans le chapitre 9 nous expliquons comment BOA gère les appels aux fonctions externes mais aussi comment nous sommes capables de détecter une reconstruction de la table d'importation d'un binaire Windows.

— Le chapitre 10 présente les solutions que nous avons implémentées afin de détecter mais aussi de gérer les exceptions au sein d'un exécutable Windows.

— Enfin, l'annexe A apporte les prérequis nécessaires à la compréhension du fonctionnement d'un programme sous une machine x86. Plus en détails, nous y décrivons les différents éléments d'une machine x86 et son fonctionnement, les étapes nécessaires au système d'exploitation afin qu'il lance l'exécution d'un fichier exécutable mais aussi le fonctionnement des exceptions dans le monde Windows.

Chapitre 2

Problématique

2.1 Désassemblage

2.1.1 Introduction

Un programme informatique peut être disponible sous plusieurs formes. De nos jours, les logiciels sont écrits dans un langage de programmation dit de « haut niveau » comme par exemple le langage C, on parle alors du code source du programme. Le fameux programme *Hello world !* est donné en langage C dans le listing 2.1. Les codes sources des programmes proposent une sémantique facilement compréhensible par

Listing 2.1 – *Hello world !* en langage C

```
#include <stdio.h>

int main() {
    printf("Hello world!");
    return 0;
}
```

les humains ce qui facilite leur compréhension même pour un développeur n'ayant pas codé le programme en question.

Cependant, les machines sont le plus souvent incapables d'exécuter un programme sous cette forme là. En effet, les processeurs ne comprennent qu'un seul langage : le langage machine. Pour que le programme *Hello world !* donné en exemple puisse être exécuté sur une machine, il est donc nécessaire de le « traduire » en langage machine, cette étape s'appelle la compilation. On obtient alors un fichier binaire que le processeur de la machine va savoir lire mais beaucoup moins compréhensible par les humains. Cependant, c'est le plus souvent sous la forme d'une fichier exécutable contenant du code écrit en langage machine que les programmes sont distribués. En effet, il n'est pas pensable que les utilisateurs finaux aient besoin de compiler eux-mêmes leur navigateur internet, par exemple, avant de pouvoir le lancer sur leur ordinateur.

L'existence de ces différents formats implique que l'activité consistant à analyser un programme est abordée différemment suivant que le programme se trouve sous la forme de son code source ou du code machine. Dans ce document, nous nous consacrons seulement au cas de l'analyse des programmes binaires exécutables, donc sous la forme du code machine.

Afin d'analyser les programmes sous cette forme, il est généralement souhaité de retrouver, à partir du fichier exécutable, les instructions assembleur correspondantes au code machine présent dans le binaire ; cette étape est appelée désassemblage. Conjointement au désassemblage, qui est un problème indécidable, il est régulier de construire, quand cela est possible, un graphe orienté appelé graphe de flot de contrôle. Ce dernier permet de représenter les différents chemins d'exécution possibles entre les instructions assembleur. Le désassemblage ainsi que la construction du graphe de flot de contrôle permettent, autant que possible,

d'effectuer le chemin inverse de la compilation et ainsi de retrouver une sémantique du programme plus exploitable que le fichier binaire en lui-même. Il est ainsi possible d'avoir une idée du code source original du programme en identifiant par exemple des fonctions dans le graphe de flot de contrôle.

Dans le cadre des logiciels malveillants par exemple, la possibilité de retrouver les instructions assembleurs et/ou le graphe de flot de contrôle du binaire est une aide précieuse afin d'étudier son fonctionnement de l'identifier comme potentiellement dangereux ou encore de permettre de le reconnaître comme une nouvelle version d'un virus déjà connu.

2.1.2 Désassemblage statique

Afin d'effectuer un désassemblage statique (désassembler sans exécuter le programme) il existe deux algorithmes bien connus. Comme l'expliquent Schwarz, Debray et Andrews [64], chacun d'entre eux a ses forces et ses faiblesses. Afin d'illustrer le fonctionnement de ces algorithmes, nous prendrons pour exemple le code machine suivant : `EB 01 68 C3 90 90 90`.

2.1.2.1 *Linear sweep*

Il s'agit de l'algorithme de désassemblage le plus simple. Il commence par lire la section de code depuis le premier octet, et il essaie de décoder chaque séquence d'octets qu'il rencontre. L'algorithme agit ainsi jusqu'à la fin de la section de code. Un des inconvénients de cette technique « brutale » est la possibilité d'obtenir des erreurs de désassemblage si le désassembleur décode des octets de donnée qu'il prend pour du code. Le désassemblage du code machine `EB0168C3909090` est donné en listing 2.2.

Listing 2.2 – Désassemblage linéaire du code x86 `EB0168C3909090`

```
0x0:  EB01          jmp 0x3
0x2:  68C3909090    push 0x909090C3
```

2.1.2.2 *Recursive disassembling*

Comparé au premier, cet algorithme se veut plus intelligent. Il commence le désassemblage à l'adresse du point d'entrée du programme, et en regardant le type de la dernière instruction désassemblée, il peut potentiellement connaître les adresses des instructions successeurs à désassembler. La faiblesse de cet algorithme apparaît lorsqu'il rencontre une instruction de type saut dynamique, dans ce cas il n'est pas toujours facile statiquement de connaître les adresses des instructions successeurs. Ainsi, il est possible que certaines parties du binaire ne soient pas désassemblées. Le désassemblage du code machine `EB0168C3909090` est donné en listing 2.3.

Listing 2.3 – Désassemblage récursif du code x86 `EB0168C3909090`

```
0x0:  EB01          jmp 0x3
0x2:  68            ;data
0x3:  C3            ret
0x4:  90            nop
0x5:  90            nop
0x6:  90            nop
```

Comme nous pouvons le remarquer dans l'exemple précédent, en utilisant simplement deux algorithmes différents nous obtenons deux désassemblages différents.

Néanmoins, les désassembleurs classiques tels que IDA [36] proposent de bonnes heuristiques de désassemblage lorsqu'ils traitent des codes binaires construits à partir de compilateurs connus. Ce genre d'outils utilise généralement une méthode de désassemblage mixant les approches *recursive* et *linear*.

2.1.3 Désassemblage hybride

Une autre façon de récupérer les instructions d'un binaire est de l'exécuter en enregistrant chaque instruction rencontrée par le processeur, on parle ici d'une analyse dynamique. À la fin de l'exécution, nous obtenons la liste des instructions de cette exécution, appelée *trace d'exécution*. Il est évident que l'exécution d'un programme dans un contexte donné ne sera pas la même pour une autre exécution dans un contexte différent. En effet, il n'existe pas nécessairement une unique trace d'exécution possible pour un binaire.

Une trace d'exécution peut également être utilisée afin d'améliorer un désassemblage statique classique. C'est notamment ce que proposent Bonfante et al. [15] dans l'outil CoDisasm en proposant un désassemblage *concatic* (*CONCrete path execution and stATIC disassembly*). Leur méthode repose sur un algorithme de désassemblage récursif qui s'appuie sur une trace d'exécution.

2.1.4 Construction du graphe de flot de contrôle

Le graphe de flot de contrôle contient toutes les informations données par le désassemblage mais l'inverse n'est pas vrai. En plus de la liste des instructions et de leurs adresses, le graphe de flot contrôle donne comme informations supplémentaires l'enchaînement possible des instructions durant l'exécution du programme. Cependant, certaines instructions utilisent des valeurs uniquement connues durant l'exécution, telle que la valeur d'un registre ou d'une case mémoire, afin de calculer l'adresse de la prochaine instruction à exécuter. Pour ces dernières, il n'est pas toujours possible de connaître statiquement les instructions successeurs, ce qui appauvrit le graphe de flot de contrôle. **Autrement dit, le fait d'avoir un désassemblage complet n'implique pas forcément la possibilité de construire un graphe de flot de contrôle complet.**

2.2 Obfuscations et techniques d'anti-analyse

Dans notre cas, une obfuscation consiste à transformer un programme P en un programme P' de façon à ce que P' fonctionne exactement comme P mais qu'il soit plus dur à comprendre [21].

Il existe un grand nombre de techniques d'obfuscations et dans ce document nous nous intéressons seulement aux obfuscations que nous traitons dans notre étude. Une obfuscation peut être appliquée sur un programme source ou directement sur un fichier exécutable.

Dans le domaine des logiciels malveillants, les techniques d'obfuscation ont pour but de rendre difficile le processus de rétro-ingénierie des fichiers binaires. L'obfuscation peut concerner les données et/ou le code exécutable du binaire. Les techniques d'obfuscation peuvent également être utilisées dans le cadre de la protection intellectuelle afin de protéger, par exemple, le code d'un logiciel propriétaire. Les obfuscations permettent aussi dans certains cas de rendre indétectable le logiciel malveillant par un antivirus.

Différentes techniques d'obfuscation existent et mettent à mal les algorithmes de désassemblage classiques.

2.2.1 Sauts dynamiques

Il existe des instructions de saut comme les `RET`, `CALL` et `JMP` pour lesquelles l'exécution est transférée à une instruction dont l'adresse dépend de l'état actuel de la machine : on appelle ce type d'instruction des sauts dynamiques comme par exemple l'instruction `JMP EAX` qui saute à l'adresse correspondante à la valeur du registre `EAX`. Ce type d'instructions est généralement utilisé de façon légitime comme par exemple pour le retour d'une fonction, lors de l'utilisation des méthodes virtuelles dans le cadre du polymorphisme, lors de l'appel d'une fonction externe ou encore avec un *switch case*. Mais elles peuvent également être utilisées pour piéger un algorithme de désassemblage récursif ou encore pour ralentir le travail de rétro-ingénierie en sautant sur une nouvelle instruction dont l'adresse est calculée à la volée. Ce type d'instructions rend le désassemblage et la construction du graphe de flot de contrôle difficile car il n'est pas possible de connaître statiquement la ou les valeurs possibles de l'adresse de saut.

L'extrait assembleur 2.4 provient du packer tElock 0.99 et montre à quel point la résolution d'un saut dynamique peut être difficile. En effet dans cet exemple, l'instruction `POP` va récupérer l'adresse de retour

précédemment sauvegardée par le `CALL`, l'instruction `INC` ajoute 1 à cette adresse, et enfin le `JMP` permet de sauter sur cette adresse. Ces trois instructions semblent simuler le comportement d'une instruction `RET` tout en falsifiant l'adresse de retour. Cet exemple de saut dynamique est cependant correctement calculé par BOA.

Listing 2.4 – Saut dynamique dans un binaire packé par tElock 0.99

```
0x100508A:  call 0x1007088    ; @[0x6ffA0] <-- 0x100508F
[...]                         ; 62 instructions plus loin
0x1006E5B:  pop ebx           ; ebx <-- @[0x6ffA0]
[...]                         ; 4 instructions plus loin
0x1006E67:  inc ebx           ; ebx <-- 0x1005090
[...]                         ; 20 instructions plus loin
0x1006E8E:  jmp ebx
```

2.2.2 Falsification de la pile d'appels

2.2.2.1 Les fonctions en assembleur

Avant de parler des techniques de falsification de la pile d'appels (*call stack tampering*), nous rappelons ici comment fonctionne le mécanisme des fonctions en assembleur.

Comme dans un langage de programmation de plus haut niveau, la notion de fonctions existe aussi en assembleur pour des raisons de factorisation du code ou encore de réutilisation d'une même partie d'un programme.

En assembleur, les deux instructions `CALL` et `RET` sont habituellement utilisées ensemble afin d'effectuer respectivement l'appel d'une fonction et le retour de la fonction au code appelant. Concrètement, l'instruction `CALL` sauvegarde en haut de la pile l'adresse de la prochaine instruction en mémoire (l'adresse de « retour » aussi appelée *return-site*) avant de transférer l'exécution à la fonction appelée. Normalement, une fois la fonction exécutée, l'instruction `RET` récupère en haut de la pile l'adresse empilée par le `CALL` afin de rendre la main à la fonction appelante.

2.2.2.2 Fonctionnement des falsifications de la pile d'appels

Le *call stack tampering*, ou falsification de la pile d'appels en français, est une technique d'obfuscation qui consiste à casser le schéma de fonctionnement habituel du duo `CALL`/`RET` tel qu'il est utilisé pour l'appel des fonctions. Cette obfuscation va faire en sorte que le `RET` ne saute pas sur l'adresse de retour précédemment poussée par le `CALL`.

Cette obfuscation engendre plusieurs problèmes lors de l'analyse d'un code binaire. Beaucoup de désassembleurs font l'hypothèse que les adresses de retour des instructions `CALL` contiennent du code. Cette hypothèse est tout à fait légitime au vu du comportement que l'on peut attendre du couple `CALL`/`RET`. Cependant, bien que cette hypothèse soit valable quand on travaille sur des binaires compilés par des compilateurs « connus », elle devient invalide dès lors que l'on travaille sur binaires malveillants utilisant des techniques de falsification de la pile d'appels. Ainsi, ces désassembleurs vont peut être essayer de désassembler des octets qui ne correspondent pas à du code exécutable. Parallèlement à ce problème, si le `RET` ne « revient » par sur l'adresse de retour du `CALL` alors cela signifie qu'il saute sur une autre adresse qui est possiblement difficile à calculer statiquement par le désassembleur. Il est alors légitime de penser que le désassembleur ne va pas forcément effectuer un désassemblage complet ou alors même effectuer un désassemblage incorrect, oubliant ainsi le code exécuté à la suite des `RET` falsifiés.

Cette obfuscation peut être réalisée de diverses façons, par exemple en modifiant explicitement sur la pile le *return-site* précédemment sauvegardé par le `CALL` ou bien encore en poussant directement une nouvelle adresse en haut de la pile avant l'exécution du `RET`.

En général, en l'absence de cette obfuscation, la hauteur de pile que l'on observe avant l'exécution du `CALL` est identique à celle que l'on a après l'exécution du `RET`. On dit que la pile reste alignée après l'exécution de la fonction. Lors d'une falsification de la pile d'appels, en fonction de la technique utilisée, il

est possible que la hauteur de pile soit différente entre l'exécution du `CALL` et celle du `RET`. Typiquement, il est possible d'altérer l'adresse *return-site* sans modifier le nombre d'éléments dans la pile, on parle alors d'une falsification de la pile d'appels alignée. Il est aussi possible de pousser explicitement une valeur sur la pile juste avant l'exécution du `RET`, dans ce cas nous avons ajouté un nouvel élément dans la pile, sa hauteur a changé, nous parlons cette fois d'une falsification de la pile d'appels désalignée.

Les binaires packés par AsPack contiennent plusieurs falsifications de la pile d'appels. La première apparition de cette obfuscation est présente dès les premières instructions de ces binaires. Prenons par exemple un binaire packé par AsPack et dont le point d'entrée est `0x1005001`. Le listing 2.5 montre le début du désassemblage de ce binaire effectué par la plateforme Radare 2.

Listing 2.5 – Exemple d'un binaire packé par AsPack : désassemblage par Radare 2

```
0x1005001:  60            pushad
0x1005002:  E803000000    call 0x100500A
0x1005007:  E9EB045D45    jmp 0x465D54F7
0x100500C:  55            push ebp
0x100500D:  C3            ret
[...]
```

Or, si nous exécutons ce binaire, nous obtenons la trace d'exécution donnée dans le listing 2.6.

Listing 2.6 – Exemple d'un binaire packé par AsPack : trace d'exécution

```
0x1005001:  pushad
0x1005002:  call 0x100500A ; push return-site 0x1005007 on the stack
0x100500A:  pop ebp        ; pop return-site address
0x100500B:  inc ebp        ; increment 0x1005007 by one
0x100500C:  push ebp       ; push on the stack the tampered return-site address
0x100500D:  ret            ; pop the value from the stack and jump on it
0x1005008:  jmp 0x100500E  ; instead of 0x1005007, we are on 0x1005008 address
```

Comme nous pouvons le remarquer, la fonction `0x100500A` effectue une falsification de la pile d'appels grâce à la séquence d'instructions `POP-INC-PUSH` et modifie ainsi l'adresse de retour utilisée par le `RET` en l'incrémentant de un. Ainsi le `RET` de la fonction `0x100500A` saute sur l'adresse `0x1005008` et non sur l'adresse `0x1005007` comme on pourrait s'y attendre.

Or, un désassembleur naïf, comme c'est ici le cas avec Radare 2, va faire la supposition que l'adresse de retour du `CALL` (ici l'adresse `0x1005007`) contient du code valide. De ce fait, il va essayer de désassembler une instruction valide à l'adresse `0x1005007`. Cette erreur conduit à un désalignement du désassemblage et donc à une erreur contrairement au désassemblage obtenu par notre plateforme BOA donné en listing 2.7. Le désassemblage proposé par Radare 2 dans le listing 2.5 est alors faux.

Listing 2.7 – Exemple d'un binaire packé par AsPack : désassemblage par BOA

```
0x1005001:  60            pushad
0x1005002:  E803000000    call 0x100500A
0x1005007:  E9            ; data
0x1005008:  EB04          jmp 0x100500E
0x100500A:  5D            pop ebp
0x100500B:  45            inc ebp
0x100500C:  55            push ebp
0x100500D:  C3            ret
```

Il est également possible d'utiliser l'instruction `RET` comme un simple saut dynamique sans prendre en considération l'éventuelle instruction `CALL` précédent le `RET`. Par exemple, à la place d'un `JMP A` pour sauter à l'adresse `A` d'un programme, il est possible d'exécuter les deux instructions `PUSH A-RET` afin de

déguiser ce saut explicite en un saut dynamique. La trace d'exécution d'un binaire packé par nPack est donnée dans le listing 2.8 et montre ce type d'obfuscation.

Listing 2.8 – Trace d'exécution d'un binaire packé par nPack 1.1.300 : saut statique déguisé

```
; @[0x1004E00] = 0x000011D7
; @[0x1004E40] = 0x01000000
0x10042CD:  mov eax, [0x1004E40]        ; move 0x1000000 in eax
0x10042D2:  mov dword [0x1004E4C], 0x1 ; not involved in the obfuscation
0x10042DC:  add [0x1004E00], eax        ; move 0x1000000 + 0x11D7 in @[0x1004E00]
0x10042E2:  push dword [0x1004E00]      ; move 0x10011D7 in @[0x6FFC0]
0x10042E8:  ret                         ; jump on @[0x6FFC0] = 0x10011D7
0x10011D7:  [...]
```

2.2.3 Prédicats opaques et branches mortes

2.2.3.1 Les instructions de saut conditionnel

Lors de son exécution, une instruction de saut conditionnel (`Jcc`) évalue une condition, par exemple, « est-ce que le dernier calcul effectué avait pour résultat zéro ? ». Si la condition est fausse, l'exécution continue de façon séquentielle et le processeur va exécuter la prochaine instruction en mémoire. Au contraire, si la condition est vraie, le processeur va transférer l'exécution à l'instruction dont l'adresse est spécifiée par l'instruction `Jcc`, on parle dans ce cas d'un « branchement ».

Pour faire la corrélation, une instruction `Jcc` est l'équivalent du *if* dans un langage de plus haut niveau.

2.2.3.2 Prédicat opaque

Au sein d'un fichier exécutable, un prédicat opaque est une technique d'obfuscation qui consiste à faire en sorte que la condition évaluée par un `Jcc` soit toujours vraie ou toujours fausse, et ce pour n'importe quelle exécution du binaire [21, 22]. De plus, la condition évaluée par le `Jcc` est généralement construite de façon à ce qu'il soit difficile pour un outil d'analyse ou un analyste de déduire qu'elle est invariable (d'où le terme « opaque »).

La conséquence directe d'un prédicat opaque est qu'une des deux branches du saut conditionnel ne pourra jamais être empruntée lors de l'exécution du binaire, on dit que cette branche est morte. Par conséquent il est possible que le code présent en aval de la branche morte ne soit jamais exécuté. On parle alors de code mort.

Cette technique permet notamment de conduire un outil de désassemblage à désassembler du code mort et à faire perdre du temps à un analyste dans une partie du binaire qui ne pourra en réalité jamais être exécutée.

Le listing 2.9 montre un exemple simple de prédicat opaque dans lequel la condition évaluée par le saut conditionnel est toujours fausse et le listing 2.10 montre un prédicat opaque que l'on peut rencontrer dans un binaire packé par eXPressor.

Listing 2.9 – Exemple d'un prédicat opaque

```
mov eax, 0x76C523AB ; 32nd bit of eax is equal to 0
xor eax, 0x12095635 ; 32nb bit of the result is equal to 0
js label            ; always false because SF always equal to 0
```

2.2.4 Auto-modification

Durant son exécution, un binaire est dit auto-modifiant s'il modifie lui-même une partie de son code avant de l'exécuter. Cette technique d'obfuscation permet à un fichier exécutable de générer du code qui

Listing 2.10 – Exemple d'un prédicat opaque dans un binaire packé par eXPressor

```
0x10058FD:  and dword ptr [ebp - 0x268], 0   ; @[X] <-- @[X] && 0 leads to @[X] <-- 0
0x1005904:  mov eax, dword ptr [ebp - 0x268] ; eax <-- 0
0x100590A:  mov dword ptr [ebp - 0x258], eax ; @[Y] <-- 0
0x1005910:  cmp dword ptr [ebp - 0x258], 0   ; compare @[Y] with 0
0x1005917:  jne 0x1005943                    ; always false (ZF always equal to 1)
```

n'existait pas au démarrage de son exécution.

Ce type d'obfuscation met à mal le désassemblage effectué par tout désassembleur statique. En effet, ces derniers n'exécutant pas le binaire qu'ils analysent, ils sont incapables de révéler les instructions que le fichier exécutable va générer lui-même au cours de son exécution. Un binaire auto-modifiant peut ainsi aisément masquer une partie de son code exécutable en faisant le nécessaire pour le générer à la volée au moment de son exécution, contraignant ainsi le travail d'un analyste.

La trace d'exécution 2.11 montre un cas simple d'auto-modification.

Listing 2.11 – Exemple construit à la main d'une auto-modification

```
0x0000:  B834120000   mov eax, 0x1234
0x0005:  BB21430000   mov ebx, 0x4321
0x000A:  FFE0         jmp eax                  ; opcode is FFE0
0x1234:  B90B000000   mov ecx, 0x000B
0x1239:  C601E3       mov byte ptr [ecx], 0xE3 ; replace @[0x000B] byte by 0xE3
0x123C:  E9C9EDFFFF   jmp 0x000A
0x000A:  FFE3         jmp ebx                  ; new opcode is FFE3
```

2.2.5 Construction de la table d'importation à la volée

Durant l'analyse d'un exécutable Windows, la lecture de la table d'importation peut révéler des indices sur le comportement que peut avoir le binaire lors de son exécution. Par exemple, si la fonction *CreateFile* est présente dans la table, il y a de fortes chances que le programme effectue des manipulations de fichiers.

Une technique couramment utilisée afin d'éviter de faire fuiter ce genre d'informations consiste à effectuer une construction de la table d'importation à la volée [20]. L'idée principale étant d'avoir le moins d'entrées possibles dans la table, et durant l'exécution, si le programme a besoin d'utiliser la fonction d'une bibliothèque externe, il s'occupera lui-même de charger la bibliothèque externe et de trouver l'adresse de la fonction dont il a besoin.

La première étape consiste à charger en mémoire les modules dont le binaire original a besoin, pour effectuer cette tâche on utilise habituellement la fonction *LoadLibrary* de l'API Windows. Cette fonction prend en paramètre le nom de la bibliothèque à charger et retourne l'adresse virtuelle à laquelle la bibliothèque a été chargée par Windows. La seconde étape consiste à récupérer les adresses virtuelles des fonctions externes dont le binaire original a besoin afin de remplir correctement la table d'importation. Pour cela la fonction *GetProcAddress* de l'API Windows est souvent utilisée. Cette fonction prend en paramètres le nom de la fonction externe pour laquelle on souhaite obtenir l'adresse ainsi que l'adresse à laquelle la bibliothèque externe accueillant cette fonction est chargée. La valeur retournée est l'adresse virtuelle de la fonction externe demandée.

Une fois ces étapes réalisées, le programme est en possession de l'adresse de la fonction externe dont il a besoin. Il peut directement l'utiliser pour faire un appel à cette fonction ou bien l'ajouter lui même à sa table d'importation.

Le listing 2.12 montre comment un binaire protégé par le packer FSG 2.0 procède afin de charger à la volée l'ensemble des DLL dont il a besoin pour fonctionner mais également comment l'adresse des fonctions externes nécessaires sont récupérées.

Listing 2.12 – Binaire packé par FSG 2.0 qui reconstruit sa table d'importation à la volée

```
; IAT entries of this binary:
;  - kernel32.dll: LoadLibraryA
;  - kernel32.dll: GetProcAddress
[...]
; eax points to 'msvcrt.dll' string
0x10001C5:  push eax
0x10001C6:  call dword [ebx+0x10] ; call LoadLibraryA with 'msvcrt.dll' argument
; Windows loads 'msvcrt.dll' DLL at 0x77BE0000 address
0x10001C9:  xchg ebp, eax         ; eax <-- 0x1, ebp <-- 0x77BE0000
0x10001CA:  mov eax, [edi]        ; eax <-- 0x10016F1
0x10001CC:  inc eax               ; eax <-- 0x10016F2
; eax points to '__p__commode' string
[...]
0x10001D4:  push eax
0x10001D5:  push ebp
0x10001D6:  call dword [ebx+0x14] ; call GetProcAddress(0x77BE0000, 0x10016F2)
; Windows retrieves address of '__p__commode' in 'msvcrt.dll' and stores it to eax
0x10001D9:  stosd                 ; save '__p__commode' address somewhere
[...]
; this procedure continues until all DLL and external functions
; are retrieved by the program
```

2.2.6 Packing

Pour mettre en place cette obfuscation, on utilise généralement des packers. Ces outils (souvent payants) permettent d'empaqueter un binaire à protéger $B_{original}$ au sein d'un binaire B_{packed}. Lors de son exécution, le binaire B_{packed} va dépaqueter le code original du binaire $B_{original}$ afin de pouvoir l'exécuter. Grâce à cette technique, il est difficile pour un simple désassembleur statique de récupérer le code source du binaire original $B_{original}$, ce dernier étant bien souvent déchiffré à la volée par le binaire B_{packed} lors de son exécution.

Cette technique est couramment utilisée afin de protéger des logiciels payants et ainsi éviter autant que possible que ces derniers soient *crackés* pour être utilisés sans licence valide. Les packers sont également souvent utilisés afin de protéger les logiciels malveillants. Cela permet parfois de faire échouer la détection par l'anti-virus, mais cela complique aussi le travail de rétro-ingénierie.

De plus, la plupart des packers ne se contentent pas seulement d'appliquer un « simple » empaquetage du binaire à camoufler, mais bien souvent ils appliquent toutes sortes de techniques d'anti-analyse au binaire final. On peut par exemple citer l'auto-modification, la falsification des piles d'appels, les sauts dynamiques, les prédicats opaques, la construction à la volée de la table d'importation et toutes sortes de techniques cherchant à détecter si le binaire est dans un environnement d'analyse afin d'arrêter son exécution prématurément.

Parmi les packers les plus connus, nous pouvons citer *tElock*, *ASPack*, *Yoda* ou encore *EP Protector*.

2.2.7 Exceptions

Quand une instruction déclenche une erreur, par exemple à cause d'une division par zéro ou bien suite à une opération en mémoire non autorisée, le processeur déclenche une exception. À la suite de cette exception, le système d'exploitation prend la main afin d'effectuer différentes étapes, la principale étant la recherche d'un gestionnaire d'exception mis à disposition par le binaire et permettant de traiter l'erreur courante. La section A.3 détaille ces différentes étapes.

Nous souhaitons ici mettre l'accent sur le fait que le déclenchement d'une exception est difficilement prévisible. Par exemple, une instruction peut faire une lecture en mémoire en utilisant la valeur d'un registre comme adresse. Statiquement, il est difficile de savoir quelle sera la valeur de ce registre lors d'une exécution et donc à quelle adresse sera effectuée la lecture. Mais surtout, il est difficile de savoir à l'avance si cette lecture va déclencher une erreur ou non.

Les exceptions ajoutent des difficultés au désassemblage, à l'exécution symbolique mais aussi aux analystes.

Un désassembleur statique naïf ne sera pas forcément en mesure de détecter quand une instruction déclenchera une exception. La première conséquence à cela est le fait que le désassembleur va possiblement rater le code du gestionnaire d'exception et ainsi oublier de désassembler du code valide du programme. La deuxième conséquence est le fait que le gestionnaire d'exception a la possibilité de ne jamais rendre la main au code qui a déclenché l'exception, ce code devient du code mort. Par la même conséquence, un analyste pourrait passer à côté de l'exception et analyser à tord une partie du code qui ne pourra jamais être exécuté.

Un gestionnaire d'exception peut aussi faire en sorte, par exemple, de modifier la valeur d'un registre avant de rendre la main au code ayant déclenché l'exception. Dans le cadre d'une exécution symbolique, si l'exception n'est pas détectée, la mauvaise valeur du registre peut fausser l'exécution symbolique.

Prenons par exemple le programme assembleur donné en listing 2.13. Lors d'une analyse statique, un analyste va chercher à savoir quelle est l'adresse de saut du `jmp eax` présent à l'adresse `0x100001C`. En regardant seulement les deux instructions séquentielles présentes en amont, l'analyste pourrait penser par erreur, que l'adresse destination du saut dynamique est `0x1006000` à cause de l'instruction `mov eax, 0x1006000`. Or si l'on exécute cet exemple à partir de l'adresse `0x1000000`, on remarque que l'instruction `div edx` va inévitablement déclencher une exception de type division par zéro. À la suite de cela, le gestionnaire d'exception d'adresse `0x1005000` va être exécuté. Ce gestionnaire va falsifier la valeur du registre `eax` dans le contexte d'exécution précédemment sauvegardé par le système d'exploitation en y plaçant la valeur `0x1007000`. Finalement, lorsque le programme reprend son exécution normalement l'instruction `jmp eax` va mener à l'adresse `0x1007000` et non à l'adresse `0x1006000` comme l'on pourrait s'y attendre. La trace d'exécution est donnée dans le listing 2.14.

Listing 2.13 – Exemple d'une technique d'anti-analyse par exception (code)

```
[...]
0x1000000:  push 0x1005000        ; commentaires dans le listing 2.14
0x1000005:  push dword [fs:0x0]
0x100000C:  mov [fs:0x0], esp
0x1000013:  xor edx, edx
0x1000015:  mov eax, 0x1006000
0x100001A:  div edx
0x100001C:  jmp eax
[...]
0x1005000:  mov ecx, [esp+0xC]
0x1005004:  mov ebx, 0x1007000
0x1005009:  mov [ecx+0xB0], ebx
0x100500F:  ret
[...]
```

Listing 2.14 – Exemple d'une technique d'anti-analyse par exception (trace d'exécution du listing 2.13)

```
[...]
0x1000000:  push 0x1005000      ; push handler address that we want to register
0x1000005:  push dword [fs:0x0] ; push previous handler record address
0x100000C:  mov [fs:0x0], esp   ; set new handler record as the first chain element
0x1000013:  xor edx, edx
0x1000015:  mov eax, 0x1006000  ; set eax to 0x1006000 value
0x100001A:  div edx             ; trigger division by zero exception
; exception dispatching procedure takes place here
; execute handler starting at address 0x1005000
0x1005000:  mov ecx, [esp+0xC]  ; get context record pointer
0x1005004:  mov ebx, 0x1007000  ; set ebx to 0x1007000 value
0x1005009:  mov [ecx+0xB0], ebx ; context[eax] <-- 0x1007000
0x100500F:  ret                 ; gives back the hand to Windows kernel
; Windows restores context_record with eax = 0x1007000
0x100001C:  jmp eax
0x1007000:  [...]
[...]
```

Chapitre 3

Sémantique

Dans ce chapitre nous définissons les concepts et objets principaux permettant le fonctionnement de BOA. Nous commençons par décrire le modèle machine utilisé au sein de notre plateforme. Ensuite nous définissons une sémantique nous permettant la prise en compte des programmes auto-modifiants ainsi que du système d'exploitation. Enfin nous définissons le graphe de flot de contrôle et le système à pile que nous utilisons conjointement afin de détecter les boucles et les falsifications de la pile d'appels.

3.1 Modèle machine

Le modèle machine que nous définissons ici est basé sur l'architecture d'une machine x86-32 bits et s'appuie sur le modèle déjà existant de la plateforme BINSEC [28]. Cependant, notre plateforme peut être adaptée sans problème pour supporter des binaires 64 bits ou encore l'architecture ARM.

3.1.1 Registres CPU et drapeaux

Dans notre modèle nous souhaitons représenter les registres CPU, cependant nous nous écartons légèrement d'une machine x86-32 stricte :

- Le registre `EIP` n'est pas représenté dans notre modèle, du moins pas sous la forme d'un registre.
- Le registre d'états `EFLAGS` n'est pas représenté en tant que tel, nous représentons seulement certains drapeaux (plus communément appelés *flags*) correspondant aux différents bits du registre d'état.
- Parmi les registres de segment, seul le registre `FS` est représenté dans notre modèle car celui-ci est régulièrement utilisé pour accéder à la structure *Thread Information Block* d'un processus Windows 32 bits.

Nous définissons les ensembles **Reg** et **Flag** correspondant respectivement aux registres CPU et aux drapeaux de notre modèle :

$$\mathbf{Reg} = \{EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, FS\}$$
$$\mathbf{Flag} = \{OF, SF, ZF, PF, CF, AF, DF, TF\}$$

Chaque registre est capable de contenir une donnée de taille 32 bits alors que les drapeaux ne peuvent prendre que pour seules valeurs 0 et 1.

La valeur des registres et des drapeaux est respectivement donnée par les applications σ_r et σ_f. Elles permettent d'associer à un registre r ou un drapeau f sa valeur v si cette dernière est concrète ou le symbole $\perp$ si elle ne l'est pas :

$$\sigma_r : r \in \mathbf{Reg} \mapsto v \in \mathbf{Addr} \cup \{\perp\}$$
$$\sigma_f : f \in \mathbf{Flag} \mapsto v \in \{0, 1, \perp\}$$

3.1.2 Mémoire

Nous définissons **Addr** $= \{0, \ldots, 2^{32} - 1\}$, l'ensemble des adresses de notre machine. Nous définissons la mémoire comme un tableau de taille 2^{32} dont chaque case (mémoire) est identifiée par une adresse $a \in$ **Addr**. Chaque case mémoire peut contenir une donnée de la taille d'un octet. L'ensemble des octets, noté **Bytes** contient l'ensemble des nombres pouvant être codés sur 8 bits.

Notre modèle mémoire est donc plat, la pile ainsi que le tas sont intégrés sans distinction particulière dans ce tableau comme le montre la figure 3.1. En comparaison au modèle mémoire utilisé par le

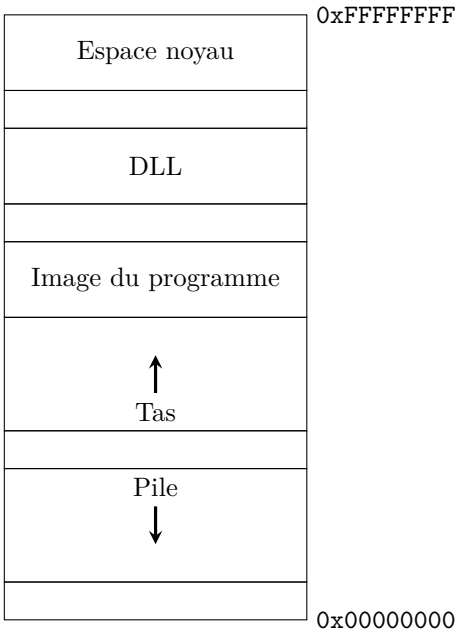


FIGURE 3.1 – Représentation simplifiée de l'espace mémoire virtuelle d'un processus Windows 32 bits

projet CompCert [47], notre approche est plus simple et nous permet de traiter correctement certaines obfuscations.

La valeur des cases mémoire est donnée par l'application σ_m. Elle permet d'associer à une case mémoire d'adresse $a \in$ **Addr** sa valeur $v \in$ **Bytes** si cette dernière est concrète ou le symbole $\perp$ si elle ne l'est pas :

$$\sigma_m : a \in \textbf{Addr} \mapsto v \in \textbf{Bytes} \cup \{\perp\}$$

En plus de sa valeur et à la différence des registres, chaque case mémoire de notre modèle se voit associer une ou plusieurs permissions d'accès. Nous notons respectivement R, W et X les droits de lecture, écriture et exécution d'une case mémoire. Les permissions associées aux cases mémoires sont données par l'application p. Elle permet d'associer à une case mémoire d'adresse $a \in$ **Addr** les droits d'accès p qui lui sont associés :

$$p : a \in \textbf{Addr} \mapsto p \in \{\emptyset, R, RX, RW, RWX\}$$

3.1.3 État machine

Durant l'exécution d'un processus et en fonction des instructions exécutées par le processeur, le pointeur d'instruction, la valeur des registres, la valeur et les permissions des cases mémoire vont changer.

Nous définissons alors un état de la machine, noté s, par un triplé $(\texttt{ip}, \sigma, p)$ composé de l'adresse $\texttt{ip} \in$ **Addr** désignant la prochaine instruction à exécuter, d'un contexte σ donnant la valeur des cases

mémoire, des registres et des drapeaux et des permissions mémoire p :

$$\texttt{ip} \quad \text{Pointeur d'instruction}$$

$$\sigma : \begin{cases} r \in \mathbf{Reg} \mapsto v \in \mathbf{Addr} \cup \{\bot\} & \text{Valeur des registres} \\ f \in \mathbf{Flag} \mapsto v \in \{0, 1, \bot\} & \text{Valeur des drapeaux} \\ a \in \mathbf{Addr} \mapsto v \in \mathbf{Bytes} \cup \{\bot\} & \text{Valeur des cases mémoire} \end{cases}$$

$$p : a \in \mathbf{Addr} \mapsto p \in \{\emptyset, R, RX, RW, RWX\} \quad \text{Permission des cases mémoire}$$

3.1.4 Notations

Pour représenter les différentes valuations d'un contexte, nous utilisons la notation « array-like ». Étant donné un contexte σ, étant donné un registre X, $\sigma_r[X]$ désigne la valeur de X dans le contexte σ et étant donné $a \in \mathbf{Addr}$, $\sigma_m[a]$ désigne l'octet présent dans la case mémoire d'adresse a.

Concernant la mémoire, $\sigma_m[\texttt{EAX}]$ représente l'octet à la position $\sigma_r[\texttt{EAX}]$ dans σ_m, qui serait normalement noté $\sigma_m\big[\sigma_r[\texttt{EAX}]\big]$, cependant la syntaxe assembleur $\sigma_m[\texttt{EAX}]$ est ici préférée.

De nombreuses instructions machines lisent des entiers (ou des réels) codés sur un plus grand nombre d'octets. C'est le cas par exemple de l'instruction `mov eax, word ptr[ebx + 4]` qui effectue une lecture en mémoire d'un mot de deux octets à partir de l'adresse $\texttt{EBX} + 4$, ce qui conduit à la lecture de la case mémoire d'adresse $\texttt{EBX} + 4$ mais aussi de la case mémoire d'adresse $(\texttt{EBX} + 4) + 1$.

Nous notons $\sigma_{m_{16}}[a]$ l'entier codé sur 16 bits à la position a en mémoire et $\sigma_{m_{32}}[a]$ celui sur 32 bits. Avec une architecture x86 qui est little-endian, $\sigma_{m_{16}}[a] = \sigma_m[a] + \sigma_m[a+1] \times 256$ et $\sigma_{m_{32}}[a] = \sigma_m[a] + \sigma_m[a+1] \times 256 + \sigma_m[a+2] \times 256^2 + \sigma_m[a+3] \times 256^3$.

Lorsque le contexte est clair, nous omettons l'indice, $\sigma_{m_{32}}[a]$ est simplifié en $\sigma_m[a]$.

De plus, nous nous laissons le droit d'utiliser la notation $\sigma[x]$ à la place de $\sigma_r[x]$, $\sigma_f[x]$ et $\sigma_m[x]$ dans le cas où x désigne respectivement un registre, un drapeau et une adresse.

3.1.5 État initial : chargement d'un fichier exécutable

Nous nous plaçons maintenant dans le cadre de l'exécution d'un fichier exécutable. Lorsque l'utilisateur souhaite démarrer l'exécution d'un fichier exécutable, il va demander au système d'exploitation de créer un processus dédié à cette exécution. Le processus comporte son propre contexte machine σ et contexte de permissions p dans lequel il va charger le binaire. De plus, le système d'exploitation se charge d'initialiser les registres ainsi que le pointeur d'instruction `ip` par l'adresse du point d'entrée du binaire notée `ep`. Aussi, le système d'exploitation charge les bibliothèques externes listées dans la table d'importation du fichier. L'état de la machine se trouve alors prêt à exécuter la première instruction du binaire.

On considère alors l'état initial de la machine noté $s_P = (\texttt{ep}, \sigma_P, p_P)$ obtenu après chargement du fichier exécutable P par le système d'exploitation. Le chargement, effectué par le système d'exploitation, comprend les étapes suivantes :

— initialisation des registres dans σ_r et des drapeaux dans σ_f ;
— copie des octets de chaque section du fichier binaire dans σ_m ;
— assignation des permissions de chaque section du fichier binaire dans p ;
— chargement des bibliothèques externes nécessaires dans σ_m ;
— assignation de `ip` par l'adresse de point d'entrée du binaire.

C'est typiquement le comportement que l'on rencontre lorsque un utilisateur Windows souhaite démarrer l'exécution d'un fichier binaire de type PE.

3.2 Sémantique

Nous formalisons dans cette section une sémantique du x86 permettant le fonctionnement de la plateforme BOA. Dans un fichier exécutable (par exemple un binaire PE), les données sont souvent entremêlées au milieu des octets du code exécutable. Il est courant dans le cas des binaires obfusqués que des octets de données se retrouvent à être interprétés comme du code exécutable. Il est alors difficile dans ce cas de

délimiter la notion de programme. Pour ces raisons, nous considérons l'interprétation d'une suite d'octets comme un programme avec une confusion complète entre code exécutable et données. Ce choix de sémantique nous permet notamment de tenir compte des problèmes d'auto-modification. De plus, nous pensons que l'exécution d'un fichier exécutable ne peut être indépendante du système d'exploitation dans lequel elle vit, pour cette raison nous représentons dans notre sémantique le système d'exploitation par une sémantique à continuation.

3.2.1 Gestion des auto-modifications

La séparation entre le code du programme d'un coté et le contexte de l'autre est un monde idéal qui ne peut pas fonctionner lorsque l'on travaille avec des binaires et en particulier lorsqu'ils sont obfusqués. Contrairement à la plupart des langages de programmation, le programme et les données ne sont pas indépendants en x86. Au début de son exécution, un fichier exécutable est chargé en mémoire par le système d'exploitation afin d'être exécuté. Durant cette exécution, il n'y a aucune garantie que le code exécutable ne soit pas modifié ou bien qu'une partie des données ne soit pas exécutée. Par exemple, durant leurs exécutions, les binaires packés génèrent à la volée de nouvelles instructions avant de les exécuter. Plus généralement, le flot de contrôle peut être dévié afin d'exécuter des données interprétées comme des opcodes d'instructions valides. Pour cela, nous ne parlons pas de « programmes x86 » mais nous utilisons plutôt la notion de « code exécutable » (une séquence d'octets).

3.2.2 Gestion de l'environnement

En sémantique des langages de programmation, le système d'exploitation est habituellement mis de coté et traité séparément. Or, nous pensons que l'exécution d'un fichier exécutable est étroitement lié au système d'exploitation. En effet, lors de son exécution, un programme interagit régulièrement avec le système d'exploitation notamment lors des appels aux fonctions de l'API ou pour la gestion des exceptions.

Dans BOA, nous souhaitons tenir compte de ces opérations au niveau noyau. D'après nous, le système d'exploitation ($\mathcal{O}$) doit être traité comme un élément à part entière dans la sémantique. En effet, une multitude d'événements comme les appels aux fonctions du noyau ou la gestion des exceptions sont gérés par le système d'exploitation qui va modifier l'état de la machine. Pour ces raisons, nous incluons l'environnement d'exécution $\mathcal{O}$ dans la sémantique [1].

Par la suite, $\mathcal{O}$ sera modélisé par un mécanisme de continuation.

3.2.3 Exécution

Étant donné un état machine $s = (\texttt{ip}, \sigma, p)$, nous définissons $\sigma[\texttt{ip}]^*$ comme la prochaine instruction `i` à exécuter. Il s'agit de l'instruction pointée par `ip` dans σ_m. L'exécution de $\sigma[\texttt{ip}]^*$ modifie $(\texttt{ip}, \sigma, p)$ menant à un nouvel état machine $(\texttt{ip}', \sigma', p')$.

Si l'exécution de $\sigma[\texttt{ip}]^*$ nécessite l'aide du noyau (par exemple dans le cas d'un appel API ou d'une exception à gérer) alors le calcul du nouvel état machine est géré par le noyau, qui, dans notre cas est représenté par l'objet $\mathcal{O}$ agissant comme un oracle. $\mathcal{O}$ prend en entrée l'état machine $(\texttt{ip}, \sigma, p)$ et calcule le nouvel état machine $\mathcal{O}(\texttt{ip}, \sigma, p)$.

Il est à noter que l'effet de modification induit par l'exécution de l'instruction `i` impacte seulement un ensemble fini des éléments de σ et de p. Par exemple, étant donné un état machine $(\texttt{ip}, \sigma, p)$, si la séquence d'octets présente en mémoire à l'adresse `ip` correspond à l'instruction x86 « `MOV EAX, EBX` »

1. Nous considérons ici que le système d'exploitation ne peut être corrompu et que le processeur ne dispose que d'un seul cœur évitant ainsi les accès mémoire concurrents.

alors l'état de la machine va évoluer vers l'état $(\mathtt{ip}', \sigma', p')$ de la façon suivante :

$$\mathtt{ip}' = \mathtt{ip} + 2 \textit{ (car l'instruction est de taille 2)}$$

$$\sigma'_r[r] = \begin{cases} \sigma_r[\mathtt{EBX}] & \text{si } r = \mathtt{EAX} \\ \sigma_r[r] & \text{sinon.} \end{cases}$$

$$\sigma'_f[f] = \sigma_f[f] \text{ pour tout drapeau } f$$

$$\sigma'_m[a] = \sigma_m[a] \text{ pour toute adresse } a$$

$$p'[a] = p[a] \text{ pour toute adresse } a$$

Finalement, notre machine x86 est modélisée par la relation binaire $\rightarrow$. La définition de $\rightarrow$ est séparée en deux cas : le cas « mode normal » lorsque le noyau $\mathcal{O}$ n'est pas nécessaire et le cas « mode noyau » sinon.

$$(\mathtt{ip}, \sigma, p) \rightarrow (\mathtt{ip}', \sigma', p') =_{def} \begin{cases} \text{Mode normal : } (\mathtt{ip}, \sigma, p) \xrightarrow{\sigma[\mathtt{ip}]^*} (\mathtt{ip}', \sigma', p') \\ \qquad \text{quand } \sigma[\mathtt{ip}]^* \text{ peut être exécuté} \\ \text{Mode noyau : } (\mathtt{ip}, \sigma, p) \xrightarrow{\mathcal{O}} \mathcal{O}(\mathtt{ip}, \sigma, p) \\ \qquad \text{sinon} \end{cases}$$

3.3 Graphe de flot de contrôle

Nous définissons un graphe de flot de contrôle (GFC) comme un graphe orienté avec une racine et dans lequel chaque nœud est un bloc de base (une suite d'instructions toutes de type séquentiel, excepté la dernière qui peut être de n'importe quel type). La racine est le nœud du GFC qui indique le point d'entrée. Formellement, un GFC est un triplet $(\mathbf{B}, \mathbf{E}, \mathtt{ep})$ où $\mathbf{B}$ est l'ensemble des nœuds (les blocs de base), $\mathbf{E}$ est l'ensemble des arcs entre les blocs de base et `ep` est la racine (le point d'entrée). Par la suite, nous allons préciser ce que nous entendons par bloc de base et préciser les propriétés attendues d'un GFC.

Il est à noter que nous avons pris le parti de définir un GFC sans faire référence explicitement à un programme. La raison est que dans le cadre de ce travail, le périmètre d'un programme n'est pas bien défini. En effet, lorsque nous avons le code source ou quand un programme est produit normalement par un compilateur, alors le périmètre du programme est bien défini. Au contraire, ici un code binaire peut cacher une partie de son code, il peut montrer du code mort ou encore avoir des branchements qui ne seront connus que lors de l'exécution. Ce dernier cas survient lors de l'exécution d'une instruction de type saut dynamique comme un `RET` ou un `JMP EAX`. De ce fait, nous avons opté pour définir un GFC par lui même, sans faire référence à un programme. Bien entendu, nous ferons le lien ultérieurement entre un code binaire et le GFC qui le représente.

3.3.1 Définition syntaxique

3.3.1.1 Les nœuds

Les blocs de base sont les nœuds des GFC et ils sont formés par une suite d'instructions qui s'enchaînent séquentiellement.

Formellement, un bloc de base, noté B, est un couple $(\mathtt{a}, \mathtt{L})$ où :

— `a` est l'adresse (appartenant à **Addr**) du point d'entrée de B ;

— `L` est la liste ordonnée des couples $\langle \textit{adresse}, \textit{instruction} \rangle$ qui composent B.

La liste `L` est composée des couples notées $\langle \mathtt{a}_0, \mathtt{i}_0 \rangle, \ldots, \langle \mathtt{a}_n, \mathtt{i}_n \rangle$ où les instructions $\mathtt{i}_0$ à $\mathtt{i}_{n-1}$ ont pour unique successeur l'instruction suivante dans le bloc de base et sont nécessairement de type séquentiel. La dernière instruction $\mathtt{i}_n$ est en général, mais pas forcément, une instruction de saut (`CALL`, `RET`, `Jcc`, `JMP`, ...).

Le point d'entrée `a` d'un bloc de base correspond à l'adresse $\mathtt{a}_0$ de l'instruction $\mathtt{i}_0$ de `L`. Pour des raisons pratiques nous notons $B_{\mathtt{a}}$ le bloc de base $(\mathtt{a}, \mathtt{L})$.

Nous pouvons noter qu'en connaissant seulement le point d'entrée d'un bloc de base et les instructions $\mathtt{i}_0$ à $\mathtt{i}_n$ présentes dans $\mathtt{L}$, il est possible de retrouver les adresses $\mathtt{a}_1$ à $\mathtt{a}_n$.

3.3.1.2 Les arcs

Un arc est un couple $(B_\mathtt{s}, B_\mathtt{d})$ qui relie le nœud $B_\mathtt{s}$ (bloc de base source) au nœud $B_\mathtt{d}$ (bloc de base destination).

Chaque nœud d'un GFC a un nombre indéfini d'arcs entrants et sortants.

3.3.2 Correction et complétude

3.3.2.1 GFC correct

Un GFC est un graphe orienté particulier dans lequel les arcs ont une notion de correction (en rapport avec la sémantique des instructions). Un arc (B, B') est dit correct si le bloc de base B' est un successeur du bloc de base B. Cela signifie que la première instruction de B' (notée $\mathtt{i}'_0$) est un successeur de la dernière instruction de B (notée $\mathtt{i}_n$). Formellement, cela signifie qu'il existe une exécution à partir du point d'entrée $\mathtt{ep}$ du GFC avec un contexte donné qui passe de $\mathtt{i}_n$ à $\mathtt{i}'_0$. Pour rappel, une instruction $\mathtt{i}'$ est dite successeur d'une instruction $\mathtt{i}$ si $\mathtt{i}'$ peut être exécutée juste après $\mathtt{i}$. Par extension, nous considérons un GFC comme correct si tous ses arcs sont corrects.

Nous faisons maintenant le lien avec la sémantique que nous avons définie dans la section 3.2. Étant donné un graphe de flot de contrôle $G = (\mathbf{B}, \mathbf{E}, \mathtt{ep})$ et un état machine $s = (\mathtt{ep}, \sigma, p)$, nous considérons que le graphe G est **correct** si pour tout chemin dans G entre la racine $B_\mathtt{ep} = (\mathtt{ep}, \mathtt{L})$ et un bloc de base $B' = (\mathtt{a}, \mathtt{L}')$ nous avons $(\mathtt{ep}, \sigma, p) \xrightarrow{*} (\mathtt{a}, \sigma', p')$ de telle façon que les opcodes des instructions $\mathtt{L}'$ appartiennent à σ' (c'est à dire que $\forall(\mathtt{a}_i, \mathtt{i}_i) \in \mathtt{L}', \mathit{opcode}(\mathtt{i}_i) \in \left[\sigma'[\mathtt{a}_i], \sigma'[\mathtt{a}_i + \mathit{size}(\mathtt{i}_i) - 1]\right]$).

Notons que les instructions de $\mathtt{L}'$ de sont pas forcément présentes dans σ. En effet, il est possible qu'elles aient été générées par auto-modification.

3.3.2.2 Chemin d'exécution

Si l'arc (B, B') est correct cela signifie que le bloc de base B' est un successeur du bloc de base B. Autrement dit, il existe une exécution à partir du point d'entrée *ep* du GFC avec un contexte donné pour laquelle B' est exécuté juste après B. Nous étendons cette notion à plusieurs blocs de base et nous appelons chemin d'exécution dans un GFC une suite d'arcs corrects. Autrement dit, un chemin d'exécution est une suite de nœuds B_0 à B_n pour lequel il existe une exécution à partir du point d'entrée *ep* du GFC avec un contexte donné permettant d'exécuter B_0, puis B_1, puis ..., et enfin B_n. Ces nœuds sont donc bien reliés un à un via des arcs corrects. Nous notons ce chemin d'exécution $B_0 \xrightarrow{*} B_n$.

3.3.2.3 GFC complet

Étant donné un fichier exécutable P et l'état initial de la machine $s_P = (\mathtt{ep}, \sigma_P, p_P)$ obtenu après le chargement de P par le système d'exploitation, nous considérons que le GFC G de P est complet si G est correct et de taille maximale en nombre d'arcs et en nombre de nœuds. Formellement, cela signifie que pour tout état machine d'entrée $s = (\mathtt{ip}, \sigma, p)$ avec $\mathtt{ip} = \mathtt{ep}$, σ_P est un sous ensemble de σ et p_P est un sous-ensemble de p, l'exécution à partir du point d'entrée $\mathtt{ep}$ du GFC est un chemin d'exécution dans le GFC. Finalement cela revient à dire que pour une exécution à partir du point d'entrée $\mathtt{ep}$, il n'existe aucun contexte d'entrée pour lequel il va manquer un arc ou un bloc de base dans le GFC.

3.4 Système à pile

Un système à pile est un 4-uplet $M = (\Gamma, Q, q_0, \Delta)$ où Γ est l'alphabet de pile contenant la pile vide notée ϵ, Q est l'ensemble des états, $q_0 \in Q$ est l'état initial et $\Delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma^*)$ est la fonction de transition. Une transition de Δ est notée $(q, \gamma) \to (q', \omega)$, cela signifie que si le système M est dans l'état q et que γ est l'élément de haut de pile, alors M peut dépiler γ, entrer dans l'état q' et empiler ω. Un cas spécial est le suivant : si $(q, \gamma) \to (q', \epsilon)$ alors γ est retiré de la pile et cette dernière est vide.

3.4.1 Les adresses *return-site*

Pour nos besoins, nous commençons par définir ce qu'est une adresse *return-site*. Étant donnée une instruction `i` d'adresse `a` et de type `CALL` (qu'il soit explicite ou dynamique), nous notons $\mathtt{a}_{return\text{-}site} = \mathtt{a} + size(\mathtt{i})$ l'adresse *return-site* de l'instruction `i`. Dans la suite nous noterons $\mathtt{a}+1$ l'adresse *return-site* du `CALL` d'adresse `a`.

Notons que nous différencions une adresse *return-site* d'une adresse de retour. La première est facilement calculable à partir d'un code désassemblé alors que la seconde désigne une adresse de retour réelle qui est dynamiquement calculée.

3.4.2 Lien avec un graphe de flot de contrôle

Un graphe de flot de contrôle $G = (\mathbf{B}, \mathbf{E}, \mathtt{ep})$ est rattaché au système à pile $M_G = (\Gamma, \mathbf{B}, B_{\mathtt{ep}}, \Delta)$ où l'alphabet de pile Γ est l'ensemble des adresses *return-site* présentes dans le graphe G et les états correspondent à l'ensemble des blocs de base de G.

Pour chaque arc (B, B') de G, il existe une transition de M_G définie de la façon suivante :

$$
\begin{aligned}
&(B,\gamma) \to (B', \alpha+1) && \text{si } B \text{ se termine par l'instruction } \alpha\texttt{: call } t\\
&(B,\gamma) \to (B', \epsilon) && \text{si } B_\gamma = B'\\
&(B,\gamma) \to (B', \gamma) && \text{sinon}
\end{aligned}
$$

Nous pouvons faire plusieurs remarques concernant ces transitions. Premièrement, dans la première règle, B' est toujours le bloc de base $B_{\mathtt{t}}$ d'adresse t. Deuxièmement, si B se termine par une instruction `ret` et que la transition qui s'applique correspond à la règle numéro 3 alors cela signifie que nous sommes en présence d'une falsification de la pile d'appels. Enfin, le système M_G est non déterministe.

Une configuration de M_G est une paire (B, π) où B est un bloc de base de G et π est la pile d'adresses *return-site*. Nous notons $(B,\pi) \to (B',\pi')$ une étape de calcul de M_G et $(B,\pi) \xrightarrow{*} (B',\pi')$ pour sa clôture transitive.

Finalement, nous appelons « pile de *return-site* » toute pile π qui vérifie $(B_{\mathtt{ep}}, \epsilon) \xrightarrow{*} (B, \pi)$.

Pour illustrer le fonctionnement d'un graphe de flot de contrôle et de son système à pile, nous prenons l'exemple du programme assembleur donné à gauche de la figure 3.2. Le graphe de flot de contrôle G correspondant à ce programme est donné en noir à droite de la figure. En rouge nous avons les différentes configurations du système M_G pour lequel le graphe G est rattaché. Enfin, en bleu sur les arcs du graphe, nous avons les différentes actions appliquées par le système. Remarquons que dans cet exemple le système M_G nous permet de caractériser l'instruction `ret` d'adresse `0x09` comme non victime d'une falsification de la pile d'appels car le bloc de base destination (`0x19`) correspond à l'adresse de haut de pile de sa configuration courante.

```
rect_perim:
    ; Args:
    ;  - eax: rectanlge length
    ;  - ebx: rectangle width
    ; Return value:
    ;  - eax: rectangle perimeter
    add eax, ebx
    mov ebx, 0x2
    mul ebx
    ret

main:
    mov eax, 0x3
    mov ebx, 0x4
    call rect_perim
    ret
```

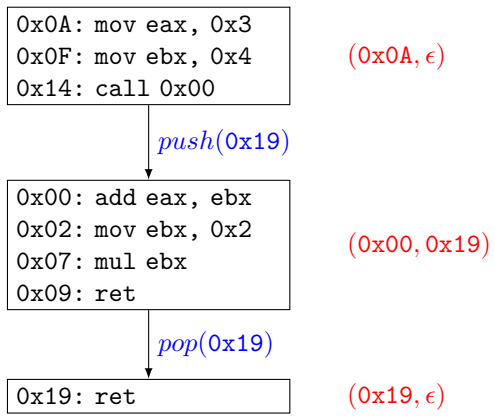


FIGURE 3.2 – Fonction sans falsification de la pile d'appels

3.4.3 Détection des boucles

3.4.3.1 Problème

Nous souhaitons pouvoir détecter les boucles présentes dans un graphe de flot de contrôle. Cette fonctionnalité est nécessaire dans BOA afin de lui permettre d'effectuer des approximations de calcul et ainsi diminuer le nombre de fois que BOA va exécuter symboliquement une boucle.

Afin de détecter la présence de boucles dans le graphe de flot de contrôle, l'idée générale que nous utilisons est d'étiqueter chaque arc par un compteur. Ainsi, durant son exécution, à chaque fois que BOA passe par un arc il incrémente son compteur. Dès que la valeur du compteur est supérieure à 1, alors cela signifie que cet arc fait possiblement parti d'une boucle. Le figure 3.3 illustre la détection d'une boucle de cette façon.

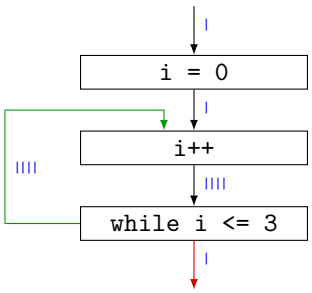


FIGURE 3.3 – Détection d'une boucle avec simples compteurs

Cependant cette technique peut détecter des boucles à tort (faux positif) comme le montre l'exemple de la figure 3.4. En effet, la fonction *f1* est appelée depuis deux endroits différents du programme (depuis les `CALL` d'adresse α et β) et BOA détecte une boucle dans la fonction *f1* (le compteur sur l'arc a atteint la valeur 2) alors que programme ne contient pas de boucle (au sens du langage assembleur). Ce cas de figure est couramment rencontré.

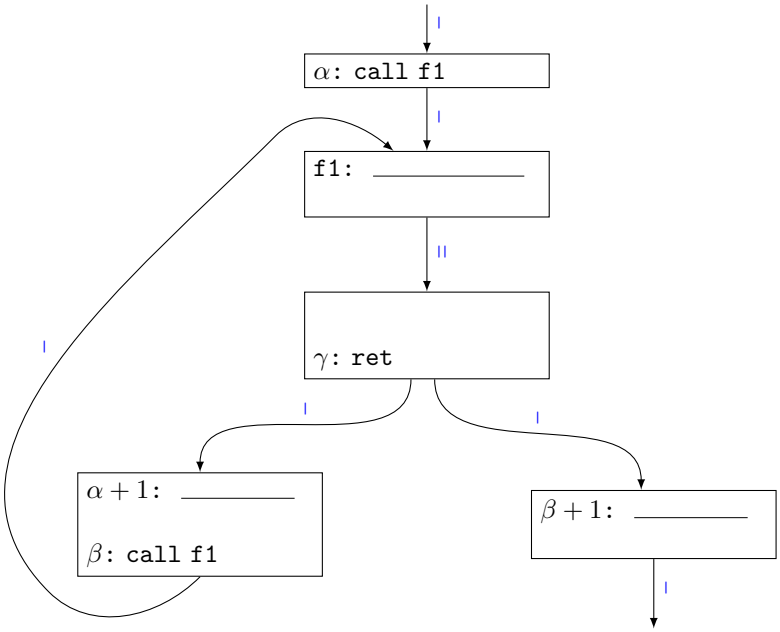


FIGURE 3.4 – Détection d'une boucle à tort avec simples compteurs

3.4.3.2 Définition des boucles

Afin de détecter les boucles dans un graphe de flot de contrôle et de résoudre le problème de faux positif décrit ci-dessus nous utilisons le système à pile défini précédemment.

Concrètement, une façon de détecter les boucles est d'utiliser plusieurs compteurs par arc, chaque compteur étant caractérisé par sa pile de *return-site*. Ainsi, quand BOA passe par un arc, il va calculer la pile de *return-site* courante grâce au système à pile et incrémenter le compteur correspondant à cette pile, on est alors capable de gérer les appels de fonction.

Finalement, nous considérons être dans une boucle lorsque nous passons deux fois sur un même arc avec la même pile de *return-site*.

La figure 3.5 montre comment cette technique permet de résoudre la détection à tort de la boucle de l'exemple précédent.

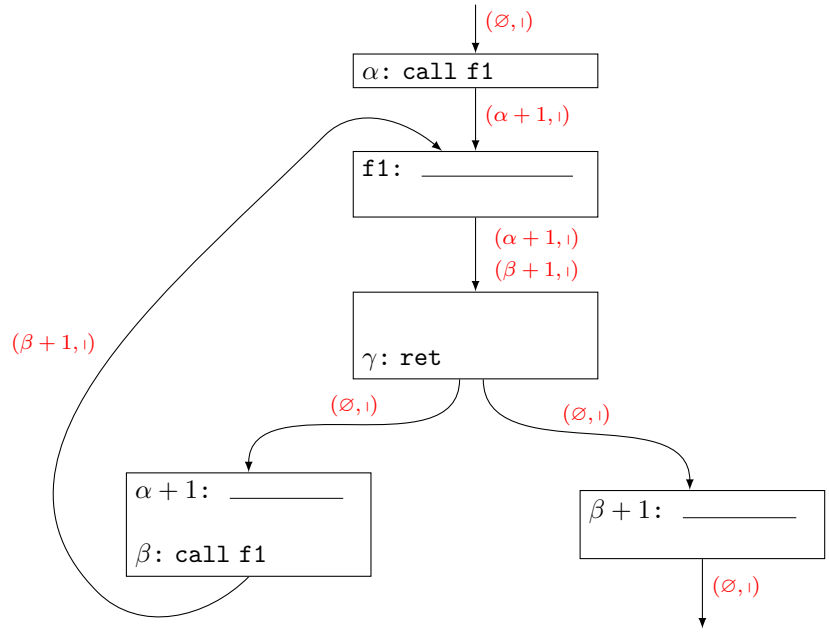


FIGURE 3.5 – Correction du faux positif de la figure 3.4

Étant donné un graphe de flot de contrôle $G = (\mathbf{B}, \mathbf{E}, \mathtt{ep})$ et le système à pile M_G auquel il est rattaché, nous détectons une boucle dans G ayant pour entrée le bloc de base B s'il existe une transition $(B_{\mathtt{ep}}, \epsilon) \xrightarrow{*} (B, \pi) \xrightarrow{*} (B, \pi)$ dans M_G (chapitre 18 du livre *Tiger book* [2]).

Finalement le chemin d'exécution $B \xrightarrow{*} B$ est une boucle dans le graphe de flot de contrôle G si la transition $(B, \pi) \xrightarrow{*} (B, \pi)$ existe dans M_G

3.4.4 Graphe de flot de contrôle étendu

Finalement, nous définissons un graphe de flot de contrôle étendu comme un GFC pour lequel les arcs sont étiquetés par une *stack closure*. Une *stack closure* ρ est une fonction de correspondance qui associe à chaque pile de *return-site* π un 4-uplet $(\mathtt{ep}, \sigma, p, n)$ où $(\mathtt{ep}, \sigma, p)$ est un état machine et n est un nombre entier qui permet de compter le nombre de fois que cet arc a été traversé avec la pile π.

3.4.4.1 Problème des appels récursifs

Nous considérons les appels récursifs comme des boucles. Cependant, à chaque fois qu'une fonction s'appelle elle-même, la pile de *return-site* change (elle augmente) et le nombre de compteurs sur les arcs augmente également. Pour autant chaque compteur reste bloqué à la valeur 1. On remarque alors qu'en

l'état, notre système de compteurs associés à une pile d'appels n'est pas capable de détecter les boucles engendrées par les appels récursifs.

Pour limiter ce problème, nous limitons le nombre d'éléments que peut contenir une pile de *return-site* permettant alors au compteur d'augmenter.

Chapitre 4

Exécution symbolique de BOA

Nous définissons dans ce chapitre la façon dont nous calculons de façon effective l'état machine s' obtenu après l'exécution d'une suite d'instructions x86 dans l'état machine s. Durant cette exécution, nous souhaitons tenir compte des auto-modifications, de l'impact des appels aux fonctions externes sur l'état de la machine mais aussi des exceptions.

Dans notre cas nous effectuons l'exécution symbolique à l'échelle du bloc de base. Pour cela nous interprétons le bloc de base B instruction par instruction, et dans l'optique d'étendre BOA à de nouvelles architectures processeurs, nous faisons le choix d'utiliser un langage intermédiaire. Nous utilisons pour cela le modèle DBA (section 4.1) défini au sein de la plateforme BINSEC [9].

Pour nos besoins, nous commençons par décrire la façon dont nous évaluons une expression DBA de façon effective (section 4.2). Cette méthode nous permet par la suite de calculer concrètement l'état machine DBA obtenu après l'exécution d'une instruction DBA (section 4.3). Finalement, nous avons tous les outils nécessaires afin de calculer les modifications engendrées sur l'état machine par l'exécution d'une instruction x86 et par composition, d'un bloc de base B (sections 4.5 et 4.4).

4.1 DBA, un langage intermédiaire

Afin d'interpréter du langage machine, nous traduisons les instructions machine en un langage intermédiaire appelé *Dynamic Bitvector Automata* (DBA). Le langage DBA a été défini dans [9] et étendu dans [29] et fait parti de la plateforme d'analyse de binaires BINSEC [2].

Une des raisons d'utiliser un langage intermédiaire comme le langage DBA qui est précis au niveau « bit » est le fait qu'il dispose d'une sémantique formelle correctement définie. De ce fait, nous sommes capables d'exécuter symboliquement une séquence d'instructions machine, à condition que celle-ci ne fasse pas intervenir le noyau. Autrement dit, il est nécessaire que cette séquence d'instructions s'exécute en « mode normal » afin de pouvoir l'exécuter symboliquement en utilisant la sémantique DBA.

Une autre raison d'utiliser un langage intermédiaire provient du fait qu'il existe différentes architectures processeurs et donc différents jeux d'instructions. Ainsi, les outils d'analyse des codes binaires doivent être capables de s'adapter aux différents jeux d'instructions. L'utilisation d'un langage intermédiaire permet d'éviter d'avoir à traiter indépendamment chaque jeu d'instructions. Le langage DBA propose un jeu d'instructions restreint capable d'encoder la sémantique des instructions d'autres jeux d'instructions comme ARM ou x86. Il est alors possible de baser le développement d'un outil d'analyse de codes binaires sur un seul langage (de représentation intermédiaire) afin qu'il soit compatible avec les différentes architectures processeurs existantes.

D'autres représentations intermédiaires existent comme BIL utilisé dans la plateforme BAP [17], VEX utilisé par angr [73], ESIL utilisé par Radare 2 ou encore REIL [31].

Il est cependant nécessaire de noter que la sémantique DBA possède plusieurs défauts pour notre utilisation. Tout d'abord le programme est fixe et ne vit pas en mémoire, ainsi, le langage DBA en tant

2. http://binsec.gforge.inria.fr/tools.html

que tel ne supporte pas les codes auto-modifiants. De plus, la sémantique DBA ne prend pas en compte l'environnement d'exécution et par conséquent la gestion des exceptions ou encore des appels systèmes.

Notons également que les instructions de type *floating-point* ne sont actuellement pas supportées par BINSEC et par le langage DBA.

4.1.1 État machine DBA

Dans le langage DBA, les valeurs appartiennent à l'ensemble des *bit vectors*, noté $\mathcal{BV}$. Un *bit vector* est un simple tableau de bits dont la taille est fixée et statiquement connue. On notera $\mathcal{BV}^n$ l'ensemble des *bit vectors* dont la valeur peut être codée sur n bits. Le langage DBA manipule un nombre fini de variables (typiquement les registres CPU). Nous notons cet ensemble Var. La taille d'une variable DBA est donnée par l'application $size : Var \to \mathbb{N}$. La mémoire, non bornée, est représentée par un tableau d'octets (chaque octet étant représenté par un *bit vector* de taille 8). Nous accédons aux éléments de la mémoire via l'opérateur @[.] en lecture ou en écriture (cela dépend de quel coté de l'instruction d'assignation l'opérateur est utilisé).

La valeur des variables est donnée par l'application δ_v. Elle permet d'associer à une variable var sa valeur $v \in \mathcal{BV}^{size(var)}$ si cette dernière est concrète ou le symbole $\perp$ si elle ne l'est pas :

$$\delta_v : var \in Var \mapsto v \in \mathcal{BV}^{size(var)} \cup \{\perp\}$$

La valeur des cases mémoire est donnée par l'application δ_m. Elle permet d'associer à une case mémoire d'adresse $a \in \mathcal{BV}^{32}$ sa valeur $v \in \mathcal{BV}^8$ si cette dernière est concrète ou le symbole $\perp$ si elle ne l'est pas :

$$\delta_m : a \in \mathcal{BV}^{32} \mapsto v \in \mathcal{BV}^8 \cup \{\perp\}$$

On notera δ, l'environnement DBA. Cette application permet de regrouper les deux applications δ_v et δ_m en un seul opérateur. Ainsi, la notation $\delta[x]$ retournera $\delta_v[x]$ si x est une variable DBA ou $\delta_m[x]$ si x est une adresse.

Les permissions associées aux cases mémoire sont données par l'application γ. Elle permet d'associer à une case mémoire d'adresse $a \in \mathcal{BV}^{32}$ les droits d'accès p qui lui sont associés :

$$\gamma : a \in \mathcal{BV}^{32} \mapsto p \in \{\emptyset, R, RX, RW, RWX\}$$

Finalement, nous définissons un état machine DBA par un triplet (ℓ, δ, γ) composé de, ℓ l'adresse de la prochaine instruction DBA à exécuter, $\delta \in \Delta$ l'environnement donnant la valeur des variables et des cases mémoire, et $\gamma \in \Gamma$ donnant la permission des cases mémoire :

$$\begin{array}{ll} \ell & \text{Pointeur d'instruction} \\ \delta : \begin{cases} var \in Var \mapsto v \in \mathcal{BV}^{size(var)} \cup \{\perp\} \\ a \in \mathcal{BV}^{32} \mapsto v \in \mathcal{BV}^8 \cup \{\perp\} \end{cases} & \begin{array}{l} \text{Valeur des variables} \\ \text{Valeur des cases mémoire} \end{array} \\ \gamma : a \in \mathcal{BV}^{32} \mapsto p \in \{\emptyset, R, RX, RW, RWX\} & \text{Permissions des cases mémoire} \end{array}$$

4.1.2 Expressions DBA

4.1.2.1 Grammaire des expressions DBA

L'ensemble des expressions DBA est noté *Expr*. Chaque expression est de taille fixe. Une expression DBA peut être une valeur $bv \in \mathcal{BV}$, une variable $v \in Var$, un élément de la mémoire ou bien la composition de ces types via des opérations logiques ou arithmétiques. De plus, une valeur spécifique, notée $\perp$ représente une valeur non définie, elle est utilisée pour certains drapeaux dont la valeur est inconnue avec certaines opérations x86.

L'ensemble des conditions est noté *Cond*, une expression conditionnelle est une expression DBA dont l'évaluation est de taille 1.

La grammaire des expressions DBA e est donnée dans la table 4.1.

bv	$\in$	$\mathcal{BV}$
v	$\in$	Var
$\Diamond_u$	$::=$	$\neg \mid -$
		$\mid ext_{u,s}(e,n)$ *(extension)*
		$\mid e\{i,j\}$ *(restriction)*
$\Diamond_b$	$::=$	$+ \mid - \mid \times \mid /_{u,s} \mid \%_{u,s} \mid = \mid \neq$
		$\mid \leq_{u,s} \mid <_{u,s} \mid \geq_{u,s} \mid >_{u,s} \mid \oplus \mid \vee \mid \wedge \mid << \mid >>_{u,s}$
		$\mid rot\_left \mid rot\_right$ *(rotation)*
		$\mid ::$ *(concaténation)*
e	$::=$	$\bot \mid bv \mid v \mid @[e] \mid \Diamond_u e \mid e \Diamond_b e$

TABLE 4.1 – Grammaire des expressions DBA

4.1.2.2 Évaluation des expressions DBA

Étant donné un environnement DBA δ et les permissions des cases mémoire γ, nous définissons la fonction $\mathcal{E} : Expr \to \Delta \to \Gamma \to \mathcal{BV} \cup \{Abort, \bot\}$ comme l'opérateur permettant d'effectuer l'évaluation d'une expression DBA. Cet opérateur est donné par les règles définies dans la table 4.2.

bv	$\in$	$\mathcal{BV}$
v	$\in$	Var
e, e_1, e_2	$\in$	$Expr$
$\mathcal{E}[\![\bot]\!]\delta$	$=$	$\bot$
$\mathcal{E}[\![bv]\!]\delta$	$=$	bv
$\mathcal{E}[\![v]\!]\delta$	$=$	$\delta_v(v)$
$\mathcal{E}[\![@[e]]\!]\delta$	$=$	$\begin{cases} \bot & \text{si } \mathcal{E}[\![e]\!]\delta = \bot \\ Abort & \text{si } \gamma(\mathcal{E}[\![e]\!]\delta) = \emptyset \quad \textit{(exception)} \\ \delta_m(\mathcal{E}[\![e]\!]\delta) & \text{sinon} \end{cases}$
$\mathcal{E}[\![\Diamond_u e]\!]\delta$	$=$	$\Diamond_u \mathcal{E}[\![e]\!]\delta$
$\mathcal{E}[\![e_1 \Diamond_b e_2]\!]\delta$	$=$	$\mathcal{E}[\![e_1]\!]\delta \Diamond_b \mathcal{E}[\![e_2]\!]\delta$

TABLE 4.2 – Règles d'évaluation d'une expression DBA

4.1.3 Instructions DBA

Les instructions DBA sont catégorisées en différents types :

- ASSIGN : Une assignation est une instruction de la forme « `lhs := expr;` ». Le membre de gauche (*lhs*) est, au choix, une variable appartenant à *Var* (ou seulement la restriction de quelques bits d'une variable), ou bien une case mémoire. Le membre de droite, *expr*, appartient à *Expr*. Cette instruction permet de mettre à jour *lhs* par la valeur de l'expression *expr*. Par exemple l'instruction DBA « A_{32}`{0, 15} := (`B_{16} `+` 4_{16}`);` » permet de mettre à jour les 16 premiers bits de la variable A par le résultat de la somme entre la variable B et le nombre 4.
- UNDEF : Ce type d'instruction est un cas particulier d'une assignation où le membre de droite est $\bot$.
- SJUMP : Le saut statique, noté « `goto l;` » permet de modifier la valeur du pointeur d'instruction par $l \in loc$ (par exemple « `goto (10000003, 0);` »).

— DJUMP : Le saut dynamique, noté « `goto expr;` » permet de modifier la valeur du pointeur d'instruction par la valeur de l'expression $expr \in Expr$ (par exemple « $\texttt{goto eax}_{32}\texttt{;}$ »).
— IF : Le saut conditionnel, noté « `if cond goto l1 else goto l2;` » permet de modifier la valeur du pointeur d'instruction par la valeur $l1$ si $cond$ vaut 1 ou $l2$ si $cond$ vaut 0 (par exemple « $\texttt{if (ZF}_1\texttt{ | (SF}_1\texttt{ != OF}_1\texttt{)) goto (2234567a, 0) else goto (100000, 1);}$ »).

Il existe d'autres types d'instructions DBA dans BINSEC qui ne sont pas donnés ici car ces types ne sont pas utilisés dans notre cas.

4.1.4 Programme DBA

Un programme DBA, noté P est composé d'une séquence d'instructions identifiées par leur adresse, $P = \ell_1 : I_1, \ldots, \ell_{m+1} : I_{m+1}$. Le domaine $supp(P)$ correspond à l'ensemble des adresses des instructions qui le compose, $\{\ell_1, \ldots, \ell_{m+1}\}$. En DBA, une adresse $\ell \in loc$ est donnée par une paire (bv, id) où $bv \in \mathcal{BV}^{32}$ correspond à l'adresse originale de l'instruction à traduire en DBA et $id \in \mathbb{N}$ est un identifiant d'adresse (dans le cas où l'instruction à traduire nécessite plusieurs instructions DBA pour être codée). Par exemple, le programme DBA correspondant à l'instruction x86 « `0x10000000: PUSH EAX` » sera

$$(0\,\text{x}10000000\,,\ 0)\colon\ @[\text{ESP}_{32} - 4_{32}]_4 := \text{EAX}_{32}\,;$$
$$(0\,\text{x}10000000\,,\ 1)\colon\ \text{ESP}_{32} := \text{ESP}_{32} - 4_{32}\,;$$

La grammaire des instructions DBA est donnée dans la table 4.3

ℓ, ℓ_1, ℓ_2	$\in$	$(\mathcal{BV} \times \mathbb{N})$
c	$\in$	$Cond$
v	$\in$	Var
e, rhs	$\in$	$Expr$
lhs	$::=$	$v \mid v\{i, j\} \mid @[e]$
i	$::=$	lhs `:=` rhs
		$\mid$ `goto` ℓ
		$\mid$ `goto` e
		$\mid$ `ite`(c)? `goto` ℓ_1; `goto` ℓ_2

TABLE 4.3 – Grammaire des instructions DBA

4.1.5 Exécution d'un programme DBA avec continuation

Notre interpréteur DBA est composé de l'environnement d'évaluation δ, des permissions mémoire γ ainsi que de la continuation de contexte $\mathcal{K}$. La continuation $\mathcal{K}$ va recevoir l'état de la machine DBA courant afin de terminer le calcul : $\mathcal{K} : (\mathcal{BV} \times \mathbb{N}) \to \Delta \to \Gamma \to \big((\mathcal{BV} \times \mathbb{N}), \Delta, \Gamma\big)$. La continuation $\mathcal{K}$ correspond à l'implémentation de $\mathcal{O}$ dans BOA.

Étant donné un programme DBA P, la transition $P \vdash_{\mathcal{K}} (\ell, \delta, \gamma) \to (\ell', \delta', \gamma')$ signifie que l'exécution de l'instruction d'adresse ℓ va mettre à jour δ vers δ', γ vers γ' et que la prochaine instruction à exécuter se trouve à l'adresse ℓ'.

Nous donnons ci-dessous les règles de transition pour chaque type d'instruction DBA :

— Si $\ell : v$ `:=` rhs

$$P \vdash_{\mathcal{K}} (\ell, \delta, \gamma) \to \mathcal{K}(\ell, \delta, \gamma) \qquad \text{si } \mathcal{E}[\![rhs]\!]\delta = Abort$$
$$P \vdash_{\mathcal{K}} (\ell, \delta, \gamma) \to (\ell + 1, \delta\big[v \leftarrow \mathcal{E}[\![rhs]\!]\delta\big], \gamma) \qquad \text{sinon}$$

— Si $\ell : v\{i,j\}$ `:=` rhs

$P \vdash_{\mathcal{K}} (\ell, \delta, \gamma) \to \mathcal{K}(\ell, \delta, \gamma)$ si $\mathcal{E}[\![rhs]\!]\delta = \mathit{Abort}$

$P \vdash_{\mathcal{K}} (\ell, \delta, \gamma) \to (\ell + 1, \delta\big[v\{i,j\} \leftarrow \mathcal{E}[\![rhs]\!]\delta\big], \gamma)$ sinon

— Si $\ell : @[e]$ `:=` rhs

$P \vdash_{\mathcal{K}} (\ell, \delta, \gamma) \to \mathcal{K}(\ell, \delta, \gamma)$ si $\mathcal{E}[\![rhs]\!]\delta = \mathit{Abort}$

$P \vdash_{\mathcal{K}} (\ell, \delta, \gamma) \to \mathcal{K}(\ell, \delta, \gamma)$ si $\mathcal{E}[\![e]\!]\delta = \mathit{Abort}$

$P \vdash_{\mathcal{K}} (\ell, \delta, \gamma) \to (\ell + 1, \delta\big[\mathcal{E}[\![e]\!]\delta \leftarrow \mathcal{E}[\![rhs]\!]\delta\big], \gamma)$ sinon

— Si $\ell :$ `goto` m avec $m \in (\mathcal{BV} \times \mathbb{N})$

$P \vdash_{\mathcal{K}} (\ell, \delta, \gamma) \to (m, \delta, \gamma)$

— Si $\ell :$ `goto` e avec $e \in \mathit{Expr}$

$P \vdash_{\mathcal{K}} (\ell, \delta, \gamma) \to \mathcal{K}(\ell, \delta, \gamma)$ si $\mathcal{E}[\![e]\!]\delta = \mathit{Abort}$

$P \vdash_{\mathcal{K}} (\ell, \delta, \gamma) \to (\mathcal{E}[\![e]\!]\delta, \delta, \gamma)$ sinon

— Si $\ell :$ `ite`(c)? `goto` ℓ_1; `goto` ℓ_2

$P \vdash_{\mathcal{K}} (\ell, \delta, \gamma) \to \mathcal{K}(\ell, \delta, \gamma)$ si $\mathcal{E}[\![c]\!]\delta = \mathit{Abort}$

$P \vdash_{\mathcal{K}} (\ell, \delta, \gamma) \to (\bot, \delta, \gamma)$ si $\mathcal{E}[\![c]\!]\delta = \bot$

$P \vdash_{\mathcal{K}} (\ell, \delta, \gamma) \to (\ell_1, \delta, \gamma)$ si $\mathcal{E}[\![c]\!]\delta = 1$

$P \vdash_{\mathcal{K}} (\ell, \delta, \gamma) \to (\ell_2, \delta, \gamma)$ si $\mathcal{E}[\![c]\!]\delta = 0$

4.2 Évaluation concrète d'une expression DBA

Dans la section 4.1.2 nous avons défini une fonction notée $\mathcal{E}$ qui permet d'évaluer une expression DBA e dans un environnement DBA δ. Ici, nous proposons un algorithme permettant d'implémenter de façon effective la fonction d'évaluation $\mathcal{E}$.

Afin de maximiser l'optimisation de cette implémentation (le temps de calcul des solveurs SMT étant relativement important), nous utiliserons une fonction de teinture que nous définirons dans cette section. L'idée générale étant de réduire autant que possible la taille des formules SMT afin de réduire le temps de calcul du solveur.

Une des principales difficultés à laquelle nous devons faire face ici, concerne les accès mémoire indirects qui compliquent la fonction de teinture. Ainsi, notre algorithme se décompose en deux parties. La première partie décrit la façon dont nous évaluons de façon effective une expression DBA dépourvue d'accès mémoire (section 4.2.1) et la seconde partie décrit l'algorithme complet permettant d'évaluer n'importe quelle expression DBA en utilisant la première partie de l'algorithme (section 4.2.2).

Nous faisons ici l'hypothèse que dans le monde DBA, lors d'un accès mémoire, l'expression utilisée comme adresse pour cette opération ne contient elle même aucun autre accès mémoire. C'est-à-dire que dans le cas d'expression DBA de la forme $@[\mathit{loc}]$ avec $\mathit{loc} \in \mathit{Expr}$, alors l'expression loc ne contient elle-même aucun accès mémoire. Dans notre cas, cette hypothèse n'a aucune influence sur nos résultats car à notre connaissance aucune instruction x86 ne permet d'effectuer un accès en mémoire à une adresse dont la valeur dépend elle même d'un autre accès mémoire.

4.2.1 Évaluation concrète d'une expression DBA sans accès mémoire

Nous définissons dans cette section la première partie de notre algorithme qui va nous permettre d'évaluer de façon effective, dans un environnement DBA δ, la valeur d'une expression DBA e ne contenant aucune opération en mémoire.

Pour cela nous commençons par définir une fonction de teinture applicable sur les expressions DBA ne contenant aucun accès mémoire (section 4.2.1.1). Ensuite nous expliquons comment calculer l'environnement DBA minimal nécessaire à l'évaluation d'une expression DBA (section 4.2.1.2). Enfin, nous expliquons comment évaluer de façon effective une expression DBA sans accès mémoire (section 4.2.1.3).

4.2.1.1 Teinture d'une expression DBA sans accès mémoire

Pour nos besoins, nous notons $\overline{Expr}$, l'ensemble des expressions DBA ne contenant aucun accès mémoire.

Nous définissons ici la fonction de teinte $\overline{\textbf{Color}} : \Delta \to \overline{Expr} \to \{True, False\}$. Cette fonction prend en paramètres un environnement DBA $\delta \in \Delta$, une expression DBA $e \in \overline{Expr}$ et retourne respectivement *True* ou *False* si la valuation de e dans δ (c'est-à-dire $\mathcal{E}[\![e]\!]\delta$) est concrète ou non (égale à $\bot$).

La table 4.4 donne les différentes règles d'évaluation de cette fonction de teinture.

bv	$\in$	$\mathcal{BV}$
v	$\in$	Var
e, e_1, e_2	$\in$	$\overline{Expr}$
$\overline{\textbf{Color}}[\![\bot]\!]\delta$	$=$	$False$
$\overline{\textbf{Color}}[\![bv]\!]\delta$	$=$	$True$
$\overline{\textbf{Color}}[\![v]\!]\delta$	$=$	$\begin{cases} False & \text{si } \delta[v] = \bot \\ True & \text{sinon} \end{cases}$
$\overline{\textbf{Color}}[\![\Diamond_u e]\!]\delta$	$=$	$\overline{\textbf{Color}}[\![e]\!]\delta$
$\overline{\textbf{Color}}[\![e_1 \Diamond_b e_2]\!]\delta$	$=$	$\overline{\textbf{Color}}[\![e_1]\!]\delta \wedge \overline{\textbf{Color}}[\![e_2]\!]\delta$

TABLE 4.4 – Règles de teinture d'une expression DBA sans accès mémoire

4.2.1.2 Calcul de l'environnement DBA minimal pour une expression sans accès mémoire

Pour rappel, nous notons $\delta = (\delta_v, \delta_m)$ l'environnement machine DBA composé des valeurs des variables (δ_v) ainsi que des valeurs des cases mémoire (δ_m).

Étant donné une expression DBA e ainsi que les variables d'un environnement DBA δ_v, nous souhaitons calculer le sous-ensemble de δ_v, noté $\delta_v \downarrow e$, qui ne contient que les variables DBA utilisées dans l'expression e.

Par exemple, si nous avons $\delta_v = \{(v_1, 3), (v_2, 5), (v_3, \bot)\}$ et $e = v_1 - v_3 + 42$, alors $\delta_v \downarrow e = \{(v_1, 3), (v_3, \bot)\}$ car la variable v_2 n'est pas utilisée dans l'expression e.

Pour effectuer ce calcul, nous commençons par récupérer l'ensemble des variables DBA utilisées dans l'expression e. Pour cela nous définissons la fonction `vars` qui prend en paramètre une expression DBA et retourne la liste des variables DBA utilisées dans cette expression. Cette fonction est définie par les règles données dans la table 4.5.

bv	$\in$	$\mathcal{BV}$
v	$\in$	Var
e, e_1, e_2	$\in$	$Expr$
$\texttt{vars}(\bot)$	$=$	$\emptyset$
$\texttt{vars}(bv)$	$=$	$\emptyset$
$\texttt{vars}(@[e])$	$=$	$\texttt{vars}(e)$
$\texttt{vars}(v)$	$=$	$\{v\}$
$\texttt{vars}(\Diamond_u e)$	$=$	$\texttt{vars}(e)$
$\texttt{vars}(e_1 \Diamond_b e_2)$	$=$	$\texttt{vars}(e_1) \cup \texttt{vars}(e_2)$

TABLE 4.5 – Règles de recherche des variables DBA présentes dans une expression DBA

Concrètement, cette fonction est implémentée au sein de la plateforme BINSEC.

Finalement nous calculons $\delta_v \downarrow e$ en réduisant le domaine de δ_v aux seules variables présentes dans $\texttt{vars}(e)$.

4.2.1.3 Définition d'un évaluateur d'une expression DBA sans accès mémoire

Finalement nous définissons la fonction $\overline{\mathbf{E}} : \Delta \rightarrow \overline{Expr} \rightarrow \mathcal{BV} \cup \{\bot\}$ qui permet d'évaluer de façon effective dans un environnement DBA δ une expression DBA e ne contenant aucun accès mémoire.

Pour cela nous procédons par étapes :

1. Nous effectuons une analyse de teinte de l'expression DBA e afin de savoir si sa valeur est concrète.
2. Si la valeur de e est concrète, nous utilisons un solveur SMT afin d'évaluer l'expression DBA e.

Le fait d'effectuer une analyse de teinte a pour nous deux avantages. Premièrement, si l'analyse de teinte nous révèle que la valeur de l'expression e n'est pas concrète alors il nous est inutile de solliciter le solveur SMT, nous faisant ainsi gagner du temps. Deuxièmement, si au contraire l'analyse de teinte nous indique que l'expression e est concrète, alors nous avons la certitude qu'une et une seule valeur est possible pour l'expression e. Ainsi, nous n'avons pas besoin d'utiliser le solveur SMT pour prouver l'unicité et nous économisons ainsi le nombre d'appels à effectuer auprès du solveur.

Étape 1 : teinture Nous commençons par effectuer une teinture de l'expression e dans l'environnement δ. Cette étape préliminaire nous permet uniquement de savoir si la valeur de l'expression e est concrète (et unique) ou inconnue.

Si nous avons $\overline{\mathbf{Color}}[\![e]\!]\delta = \mathit{False}$ alors cela signifie que la valeur de l'expression DBA e dans l'environnement δ est inconnue. Il n'est alors pas nécessaire de faire appel au solveur SMT et nous pouvons directement conclure que $\overline{\mathbf{E}}[\![e]\!]\delta = \bot$. Dans le cas contraire, si $\overline{\mathbf{Color}}[\![e]\!]\delta = \mathit{True}$, il est nécessaire d'utiliser un solveur SMT afin d'évaluer l'expression DBA e. Remarquons que si $\overline{\mathbf{Color}}[\![e]\!]\delta = \mathit{True}$, alors $\forall(var, val) \in \delta_v \downarrow e$ nous avons $val \neq \bot$.

Étape 2 : évaluation par un solveur SMT Durant cette étape, nous savons déjà que $\overline{\mathbf{Color}}[\![e]\!]\delta = \mathit{True}$, cela signifie que l'expression DBA e dans l'environnement DBA δ évalue à une et une seule valeur concrète. Pour la calculer, nous utilisons un solveur SMT. Nous commençons par créer une formule SMT ϕ dont la sémantique est équivalente à l'expression DBA e. Pour cela, nous utilisons la plateforme BINSEC qui permet de traduite n'importe quelle expression DBA en une formule au format SMTLIB [11] sémantiquement équivalente.

Ensuite, nous récupérons l'environnement minimal des variables $\delta_v \downarrow e$ grâce à la procédure que nous avons définie dans la section 4.2.1.2. Pour chaque paire $(var, val) \in \delta_v \downarrow e$, nous ajoutons dans la formule ϕ une contrainte $var = val$.

Finalement, nous faisons évaluer la formule ϕ par un solveur SMT et nous demandons au solveur qu'il nous retourne la valeur v_{SAT} correspondante à l'évaluation de l'expression e. Nous avons alors $\overline{\mathbf{E}}[\![e]\!]\delta = v_{SAT}$.

Remarque d'implémentation Dans la section 4.2.1.2 nous expliquons comment, à partir d'une expression DBA e, nous réduisons l'environnement des variables DBA afin de ne conserver que les informations utiles à l'évaluation de l'expression e. Le seul but de cette procédure est d'« alléger » l'état initial imposé au solveur SMT. En théorie nous pourrions, lors de la construction de la formule SMT ϕ, ajouter sous la forme de contraintes l'ensemble des informations présentes dans δ_v. Cependant cela augmenterait le temps nécessaire au solveur SMT pour effectuer l'évaluation de l'expression e et pour autant cela ne changerait rien au résultat final.

4.2.2 Évaluation concrète d'une expression DBA avec accès mémoire

Nous souhaitons maintenant définir une fonction capable d'évaluer de façon effective n'importe quelle expression DBA (sous entendu, avec ou sans accès mémoire).

Pour cela, nous commençons par étendre la fonction de teinte donnée dans la section 4.2.1.1 afin qu'elle soit capable de teinter une expression DBA contenant des opérations mémoire. La définition de la fonction de teinture « complète » (section 4.2.2.1) est rendue possible grâce à l'évaluateur $\overline{\mathbf{E}}$ défini précédemment.

Ensuite, nous expliquons comment calculer l'environnement DBA minimal d'une expression DBA e, c'est-à-dire l'environnement DBA réduit contenant seulement les informations des variables et cases mémoire utiles à l'évaluation de l'expression e (section 4.2.2.2).

Enfin, nous pouvons définir $\mathbf{E}$, l'opérateur nous permettant d'évaluer de façon effective une expression DBA e dans un environnement DBA δ (section 4.2.2.3).

4.2.2.1 Teinture d'une expression DBA

Nous étendons ici la fonction de teinture définie dans la section 4.2.1.1 afin d'ajouter le support des accès mémoire et ainsi teinter n'importe quelle expression DBA. Nous notons cette fonction $\mathbf{Color} : \Delta \to \mathit{Expr} \to \{\mathit{True}, \mathit{False}\}$. C'est grâce à l'implémentation de la fonction $\overline{\mathbf{E}}$ que nous pouvons complètement définir la fonction de teinture.

La fonction **Color** prend en paramètre un environnement DBA $\delta \in \Delta$, une expression DBA $e \in \mathit{Expr}$ et retourne respectivement *False* ou *True* si la valuation de e dans δ a pour valeur $\bot$ ou non.

La table 4.6 donne les différentes règles d'évaluation de cette fonction de teinture.

bv	$\in$	$\mathcal{BV}$
v	$\in$	Var
e, e_1, e_2	$\in$	Expr
$\mathbf{Color}[\![\bot]\!]\delta$	$=$	False
$\mathbf{Color}[\![bv]\!]\delta$	$=$	True
$\mathbf{Color}[\![v]\!]\delta$	$=$	$\begin{cases} \mathit{False} & \text{si } \delta[v] = \bot \\ \mathit{True} & \text{sinon} \end{cases}$
$\mathbf{Color}[\![@[e]]\!]\delta$	$=$	$\begin{cases} \mathit{False} & \text{si } \mathbf{Color}[\![e]\!]\delta = \mathit{False} \\ \mathit{False} & \text{si } \delta\left[\overline{\mathbf{E}}[\![e]\!]\delta\right] = \bot \\ \mathit{True} & \text{sinon} \end{cases}$
$\mathbf{Color}[\![\Diamond_u e]\!]\delta$	$=$	$\mathbf{Color}[\![e]\!]\delta$
$\mathbf{Color}[\![e_1 \Diamond_b e_2]\!]\delta$	$=$	$\mathbf{Color}[\![e_1]\!]\delta \wedge \mathbf{Color}[\![e_2]\!]\delta$

TABLE 4.6 – Règles de teinture d'une expression DBA

4.2.2.2 Calcul de l'environnement minimal pour une expression DBA

Étant donné une expression DBA e et un environnement DBA δ, nous souhaitons calculer $\delta \downarrow e$, l'environnement DBA minimal utile à l'évaluation de l'expression e. Comme $\delta \downarrow e = (\delta_v \downarrow e, \delta_m \downarrow e)$, cela revient à effectuer le calcul en deux temps avec en premier le calcul de $\delta_v \downarrow e$ et enfin le calcul de $\delta_m \downarrow e$. De plus, nous avons déjà donné dans la section 4.2.1.2 la méthode permettant de calculer $\delta_v \downarrow e$.

Nous devons maintenant calculer $\delta_m \downarrow e$, le sous-ensemble des valeurs mémoire utiles à l'expression e. Cependant, à cause des accès mémoire indirects, la démarche est légèrement plus compliquée que pour les variables. En effet, une étape supplémentaire est nécessaire afin de calculer la valeur concrète des adresses utilisées pour les différents accès mémoire effectués dans l'expression DBA e.

Ainsi, pour calculer $\delta_m \downarrow e$, une étape préliminaire est nécessaire afin de rechercher dans l'expression DBA e, l'ensemble des expressions DBA utilisées comme adresses des opérations mémoire effectuées dans e. Une fois en possession de cette liste d'expressions DBA, nous pouvons utiliser l'opérateur $\overline{\mathbf{E}}$ afin

d'obtenir la valeur concrète des adresses des opérations mémoire effectuées dans l'expression DBA e. Enfin, nous pouvons calculer $\delta_m \downarrow e$.

Nous donnons ici la méthode que nous utilisons pour calculer $\delta_m \downarrow e$ afin de pouvoir finalement construire $\delta \downarrow e = (\delta_v \downarrow e, \delta_m \downarrow e)$.

Étape 1 : recherche des expressions DBA utilisées comme adresses des accès mémoire Nous commençons par récupérer l'ensemble des « sous-expressions » DBA utilisées comme adresses des accès mémoire effectués dans e. Nous notons `locs` la fonction qui prend en paramètre une expression DBA e et retourne la liste des expressions DBA utilisées comme adresses des opérations mémoire dans l'expression e. Cette fonction est définie par les règles données dans la table 4.7.

bv	$\in$	$\mathcal{BV}$
v	$\in$	Var
e, e_1, e_2	$\in$	$Expr$
$\texttt{locs}(\bot)$	$=$	$\emptyset$
$\texttt{locs}(bv)$	$=$	$\emptyset$
$\texttt{locs}(@[e])$	$=$	$\{e\}$
$\texttt{locs}(v)$	$=$	$\emptyset$
$\texttt{locs}(\Diamond_u e)$	$=$	$\texttt{locs}(e)$
$\texttt{locs}(e_1 \Diamond_b e_2)$	$=$	$\texttt{locs}(e_1) \cup \texttt{locs}(e_2)$

TABLE 4.7 – Règles de recherche des expressions DBA utilisées comme adresses dans une expression DBA

Concrètement, cette fonction est implémentée au sein de la plateforme BINSEC.

Finalement, nous récupérons $\texttt{locs}(e)$ la liste des expressions DBA utilisées comme adresses dans l'expression DBA e.

Étape 2 : évaluation effective des adresses mémoire Étant donné une expression DBA e et la liste $\texttt{locs}(e)$, nous souhaitons calculer grâce à l'opérateur $\overline{\mathbf{E}}$ la valeur concrète des adresses correspondantes aux expressions de $\texttt{locs}(e)$ (quand cela est possible). Nous construisons alors une nouvelle liste notée $\texttt{locs}(e)^c$ qui contiendra les valeurs concrètes des expressions présentes dans $\texttt{locs}(e)$. Ainsi, pour chaque expression $loc \in \texttt{locs}(e)$, si $\overline{\mathbf{E}}[\![loc]\!]\delta \neq \bot$ alors $\overline{\mathbf{E}}[\![loc]\!]\delta$ est ajouté dans $\texttt{locs}(e)^c$.

Étape 3 : construction de l'environnement minimal des cases mémoire $\delta_m \downarrow e$ Finalement nous calculons $\delta_m \downarrow e$ en réduisant le domaine de δ_m aux seules adresses présentes dans $\texttt{locs}(e)^c$.

Nous souhaitons rappeler que dans le cas d'un accès mémoire deux « niveaux » sont à prendre en compte. Le premier niveau correspond à l'adresse où est effectuée l'opération en mémoire, dans le cas d'un accès mémoire indirect il est possible que la valeur de cette adresse ne soit pas concrète (égale à $\bot$). Dans ce cas, il est impossible de savoir sur quelle case mémoire l'opération de lecture ou d'écriture est effectuée. Le second niveau, dans le cas où la valeur de l'adresse mémoire est concrète, correspond à la case mémoire en elle-même. Dans ce cas, nous savons sur quelle case mémoire est effectuée l'opération, cependant dans le cas d'une opération de lecture par exemple, il est possible que la valeur présente dans la case mémoire concernée soit non concrète. Autrement dit, pour connaître la valeur récupérée par une lecture mémoire il est nécessaire de (i) connaître l'adresse de la lecture et (ii) que la valeur dans cette case mémoire soit concrète dans δ_m.

Calcul de $\delta \downarrow e$ Finalement nous sommes en mesure de calculer $\delta \downarrow e = (\delta_v \downarrow e, \delta_m \downarrow e)$, l'environnement DBA minimal de l'expression e grâce à $\delta_v \downarrow e$ défini dans la section 4.2.1.2 et grâce au calcul de $\delta_m \downarrow e$ donné ci-dessus.

4.2.2.3 Définition d'un évaluateur d'une expression DBA

Nous avons maintenant l'ensemble des outils nécessaires afin de définir l'évaluateur $\mathbf{E} : \Delta \rightarrow \mathit{Expr} \rightarrow \mathcal{BV} \cup \{\bot\}$ qui permet d'évaluer n'importe quelle expression DBA e dans un environnement DBA δ. Cette fonction est une extension de l'évaluateur $\overline{\mathbf{E}}$ défini dans la section 4.2.1 qui permet d'évaluer de façon effective une expression DBA dépourvue d'accès mémoire.

Pour cela nous procédons par étapes :

1. Nous effectuons une analyse de teinte de l'expression DBA e afin de savoir si sa valeur est concrète.
2. Si la valeur de e est concrète, nous utilisons un solveur SMT afin d'évaluer l'expression DBA e.

Étape 1 : teinture Dans un premier temps, nous cherchons à savoir si l'expression DBA e évalue à une valeur concrète dans l'environnement DBA δ. Pour cela, nous utilisons l'opérateur de teinture précédemment défini et nous calculons $\mathbf{Color}[\![e]\!]\delta$.

Si $\mathbf{Color}[\![e]\!]\delta = \mathit{False}$ alors la valeur de e n'est pas concrète. Dans ce cas, il est inutile de solliciter le solveur SMT, nous pouvons directement conclure que $\mathbf{E}[\![e]\!]\delta = \bot$.

Sinon si $\mathbf{Color}[\![e]\!]\delta = \mathit{True}$ cela signifie que la valeur de e dans l'environnement δ est concrète. Ce résultat nous permet également d'affirmer que les propriétés suivantes sont respectées :

1. $\forall (\mathit{var}, \mathit{val}) \in \delta_v \downarrow e$ nous avons $\mathit{val} \neq \bot$ (toutes les variables utilisées dans l'expression e sont concrètes dans l'environnement DBA δ)
2. $\forall \mathit{loc} \in \texttt{locs}(e)$ nous avons $\overline{\mathbf{E}}[\![\mathit{loc}]\!]\delta \neq \bot$ (pour chaque opération en mémoire effectuée dans l'expression e, la valeur de l'adresse mémoire de l'opération est concrète)
3. $\forall (\mathit{addr}, \mathit{val}) \in \delta_m \downarrow e$ nous avons $\mathit{val} \neq \bot$ (pour chaque opération en mémoire effectuée dans l'expression e, la valeur lue ou écrite en mémoire est concrète)

Comme la teinte de l'expression e indique que sa valeur est concrète alors nous passons à l'étape 2 afin de calculer cette valeur grâce au solveur SMT.

Étape 2 : évaluation par un solveur SMT Maintenant que nous savons que l'expression DBA e est concrète, nous utilisons un solveur SMT afin de calculer sa valeur.

Pour cela, nous commençons par créer une formule SMT ϕ dont la sémantique est équivalente à l'expression DBA e.

Ensuite, nous calculons l'environnement DBA minimal $\delta \downarrow e$ grâce à la procédure que nous avons définie précédemment. Pour chaque paire $(\mathit{var}, \mathit{val}) \in \delta_v \downarrow e$, nous ajoutons dans la formule ϕ une contrainte $\mathit{var} = \mathit{val}$. De la même façon, pour chaque paire $(\mathit{addr}, \mathit{val}) \in \delta_m \downarrow e$, nous ajoutons dans la formule ϕ une contrainte $@[\mathit{addr}] = \mathit{val}$.

Finalement, nous faisons évaluer la formule ϕ par un solveur SMT et nous demandons au solveur qu'il nous retourne la valeur v_{SAT} correspondante à l'évaluation de l'expression e. Nous avons alors $\mathbf{E}[\![e]\!]\delta = v_{SAT}$.

L'algorithme 1 résume l'implémentation de l'opérateur $\mathbf{E}$ permettant l'évaluation d'une expression DBA.

4.3 Exécution symbolique d'une instruction DBA

Étant donnée une instruction DBA I d'adresse ℓ, un environnement DBA δ et les permissions mémoire γ, nous souhaitons calculer de façon effective l'état machine $(\ell', \delta', \gamma')$ obtenu après l'exécution de l'instruction I dans l'état machine (ℓ, δ, γ).

Pour cela, nous utilisons l'opérateur d'évaluation des expressions DBA $\mathbf{E}$ que nous venons de définir. Nous rappelons également que nous devons faire le nécessaire afin de gérer les cas particuliers faisant intervenir la continuation $\mathcal{K}$ (gestion des erreurs et des appels systèmes, voir section 4.1.5). Concrètement, BOA sera utilisé afin de finir le calcul de l'état machine lorsque la continuation $\mathcal{K}$ est nécessaire. De façon générique, nous notons $\mathcal{BOA}$ la fonction dans BOA chargée d'implémenter la continuation $\mathcal{K}$ mais le fonctionnement et l'implémentation des différents cas sont traités indépendamment dans le chapitre 10 pour la gestion des exceptions et la section 9.1 pour la gestion des appels systèmes.

Algorithme 1 : Évaluation effective d'une expression DBA

Function `EvalDBAExpr(`δ`,` e`)` **:**
- **Input :** L'environnement DBA δ dans lequel faire l'évaluation
  L'expression DBA e à évaluer
- **Output :** La valeur de l'évaluation de l'expression DBA e
- **if Color**$[\![e]\!]\delta$ **then**
  - `/* La valeur de` e `est concrète */`
  - $\phi = [\,]$
  - **foreach** $(var, val) \in \delta_v \downarrow e$ **do**
    - $\phi = \phi +$ `SMTConstraint(`var`,` val`)`
  - **foreach** $(addr, val) \in \delta_m \downarrow e$ **do**
    - $\phi = \phi +$ `SMTConstraint(`@$[addr]$`,` val`)`
  - **return** `SMTGetValue(`ϕ`,` e`)`
- **else**
  - `/* La valeur de` e `n'est pas concrète */`
  - **return** $\bot$

Finalement, nous procédons différemment en fonction du type de l'instruction DBA I à exécuter.

4.3.1 Assignation

Si l'instruction DBA I est une assignation alors nous procédons différemment en fonction de sa forme.
Si I est de la forme v`:=`rhs alors l'état machine DBA évolue de la façon suivante :

$$\ell' = \ell + 1$$
$$\delta'_v[var] = \begin{cases} \mathbf{E}[\![rhs]\!]\delta & \text{si } var = v \\ \delta_v[var] & \text{sinon} \end{cases}$$
$$\delta'_m = \delta_m$$
$$\gamma' = \gamma$$

Si I est de la forme $v\{i, j\}$`:=`rhs alors l'état machine DBA évolue de la façon suivante :

$$\ell' = \ell + 1$$
$$\delta'_v[var] = \begin{cases} \bot & \text{si } var = v \text{ et } \mathbf{E}[\![e]\!]\delta = \bot \\ \delta_v[var]\{0, i-1\} :: \mathbf{E}[\![e]\!]\delta :: \delta_v[var]\{j+1, size(var)-1\} & \text{si } var = v \text{ et } \mathbf{E}[\![e]\!]\delta \neq \bot \\ \delta_v[var] & \text{sinon} \end{cases}$$
$$\delta'_m = \delta_m$$
$$\gamma' = \gamma$$

Si I est de la forme @$[loc]$`:=`rhs alors nous devons d'abord calculer l'adresse de la case mémoire qui est modifiée. Pour cela nous calculons $\mathbf{E}[\![loc]\!]\delta$.

Si $\mathbf{E}[\![loc]\!]\delta = \bot$ alors nous ne savons pas quelle case mémoire est modifiée par cette instruction DBA. Dans ce cas nous faisons l'approximation disant que l'état de la machine n'a pas changé et nous avons $(\ell', \delta', \gamma') = (\ell, \delta, \gamma)$.

Sinon, si nous savons quelle case mémoire est écrite (c'est-à-dire que $\mathbf{E}[\![loc]\!]\delta = addr$ avec $addr \neq \bot$)

alors l'état de la machine DBA évolue de la façon suivante :

$$\ell' = \ell + 1$$
$$\delta'_v = \delta_v$$
$$\delta'_m[a] = \begin{cases} \mathbf{E}[\![rhs]\!]\delta & \text{si } a = \mathbf{E}[\![loc]\!]\delta \\ \delta_m[a] & \text{sinon} \end{cases}$$
$$\gamma' = \gamma$$

4.3.2 Saut statique

Si l'instruction DBA I est de type saut statique (de la forme `goto` m avec $m \in (\mathcal{BV} \times \mathbb{N})$) alors l'état machine DBA évolue de la façon suivante :

$$\ell' = m$$
$$\delta'_v = \delta_v$$
$$\delta'_m = \delta_m$$
$$\gamma' = \gamma$$

4.3.3 Saut dynamique

Si l'instruction DBA I est de type saut dynamique (de la forme `goto` *loc* avec *loc* $\in$ *Expr*) alors l'état machine DBA évolue de la façon suivante :

$$\ell' = \mathbf{E}[\![loc]\!]\delta$$
$$\delta'_v = \delta_v$$
$$\delta'_m = \delta_m$$
$$\gamma' = \gamma$$

4.3.4 Saut conditionnel

Si l'instruction DBA I est un saut conditionnel (de la forme `ite`(c)? `goto` ℓ_1; `goto` ℓ_2) alors nous évaluons la condition c afin de connaître l'adresse de la prochaine instruction à exécuter.

L'état machine DBA évolue de la façon suivante :

$$\ell' = \begin{cases} \ell_1 & \text{si } \mathbf{E}[\![c]\!]\delta = 1 \\ \ell_0 & \text{si } \mathbf{E}[\![c]\!]\delta = 0 \\ \bot & \text{sinon} \end{cases}$$
$$\delta'_v = \delta_v$$
$$\delta'_m = \delta_m$$
$$\gamma' = \gamma$$

4.4 Traduction x86 et DBA

Dans cette section, nous décrivons comment, étant donné un état de la machine x86 $(\texttt{ip}, \sigma, p)$, nous calculons, en utilisant la sémantique DBA, le prochain état machine $(\texttt{ip}', \sigma', p')$ après exécution potentielle de l'instruction `i` présente à l'adresse `ip` dans le contexte σ.

4.4.1 Traduction d'une instruction x86 en un programme DBA

La plateforme BINSEC fournit une fonction de traduction notée $\mathcal{T}_{inst}$ qui permet de compiler une instruction x86 `inst` en un programme DBA sémantiquement équivalent $\mathcal{T}_{inst}(\texttt{inst})$.

Pour une instruction x86 `inst` d'adresse a, toutes les instructions DBA du programme $\mathcal{T}_{inst}(\texttt{inst})$ ont une adresse de la forme (a, i) avec $i \in [0, n-1]$ s'il faut n instructions DBA pour traduire l'instruction `inst`. De plus, le point d'entrée du programme $\mathcal{T}_{inst}(\texttt{inst})$ sera la première instruction d'adresse $(a, 0)$.

4.4.2 Traduction d'un environnement machine x86 en DBA

La plateforme BINSEC fournit également une fonction de traduction notée $\mathcal{T}_{env}$ qui permet de traduire un état machine x86 $(\texttt{ip}, \sigma, p)$ en un état DBA $(\ell, \delta, \gamma) = \mathcal{T}_{env}(\texttt{ip}, \sigma, p)$. Cette traduction est immédiate et sans perte car les éléments des états machines DBA et x86 sont de même nature.

4.4.3 Sémantique du x86 via les DBA

La correction des fonctions $\mathcal{T}_{inst}$ et $\mathcal{T}_{env}$ est garantie par BINSEC lorsque l'exécution en x86 est en « mode normal », sous entendu lorsque le noyau n'intervient pas. Dans notre cas, nous garantissons la correction grâce à la continuation $\mathcal{K}$ qui permet de gérer les cas particuliers. Ceci étant dit, nous obtenons la propriété suivante :

Si

$$(\texttt{ip}, \sigma, p) \to (\texttt{ip}', \sigma', p')$$

alors

$$\mathcal{T}_{inst}(\sigma[\texttt{ip}]^*) \vdash_{\mathcal{K}} \mathcal{T}_{env}(\texttt{ip}, \sigma, p) \to (\ell', \delta', \gamma')$$

et

$$\mathcal{T}_{env}(\texttt{ip}', \sigma', p') = (\ell', \delta', \gamma')$$

Cependant, cette propriété n'est pas composable car l'exécution d'une instruction x86 peut appliquer une auto-modification d'une autre instruction x86, créant alors de nouvelles instructions x86 à la volée. Dans ce cas, l'opcode d'une instruction initialement présente dans σ est modifiée et enregistrée dans σ'. Cela signifie que cette propriété fonctionne pour une instruction x86, mais pour exécuter la prochaine instruction x86, il est nécessaire de générer un nouveau programme DBA. De ce fait, nous devons traduire la prochaine instruction x86 $\sigma'[\texttt{ip}']^*$ en un nouveau programme DBA $\mathcal{T}_{inst}(\sigma'[\texttt{ip}']^*)$.

Finalement, l'interprétation d'un code binaire x86 en DBA est décrit dans la figure 4.1 montrant ainsi qu'à chaque étape un nouveau programme DBA est traduit et interprété. Ce mécanisme est un processus réflexif qui revient à traduire de la donnée en programmes.

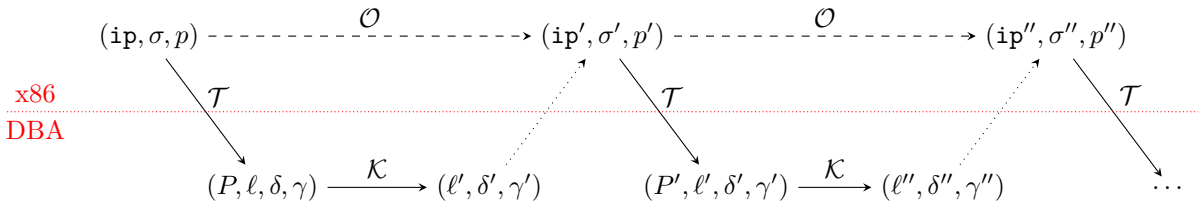


FIGURE 4.1 – Interprétation d'un code binaire x86 via les DBA

Nous donnons ci-dessous les étapes détaillées nécessaires afin d'interpréter le code binaire x86 à partir de l'état machine $(\texttt{ip}, \sigma, p)$.

4.4.3.1 Étape 1 : Traduction de l'instruction x86 en un programme DBA

Nous commençons par traduire l'instruction x86 présente à l'adresse `ip` dans σ en un programme DBA sémantiquement équivalent :

$$P = \mathcal{T}_{inst}(\sigma[\texttt{ip}]^*)$$

4.4.3.2 Étape 2 : Traduction de l'état machine x86 en DBA

Nous utilisons BINSEC afin de traduire l'état machine x86 en un état machine DBA équivalent :

$$(\ell, \delta, \gamma) = \mathcal{T}_{env}(\texttt{ip}, \sigma, p)$$

4.4.3.3 Étape 3 : Interprétation du programme DBA P

Finalement nous utilisons l'interpréteur DBA accompagné de la continuation $\mathcal{K}$ afin d'exécuter le programme DBA P dans l'état machine (ℓ, δ, γ) :

$$P \vdash_{\mathcal{K}} (\ell, \delta, \gamma) \rightarrow (\ell', \delta', \gamma')$$

À partir de là, deux cas sont possibles :
Si $\ell' = \bot$ alors l'adresse de la prochaine instruction à exécuter est inconnue. L'exécution symbolique du code binaire x86 est alors terminée.
Sinon si $\ell' \neq \bot$ alors il est nécessaire de construire un nouveau programme DBA afin de continuer l'exécution du code binaire x86. Nous savons qu'il existe un état machine x86 $(\texttt{ip}', \sigma', p')$ tel que $\mathcal{T}_{env}(\texttt{ip}', \sigma', p') = (\ell', \delta', \gamma')$. Nous pouvons alors, à partir de ce nouvel état machine x86, revenir à l'étape 1.

4.5 Exécution symbolique d'une suite d'instructions x86

Nous souhaitons maintenant calculer l'état machine s' obtenu après exécution d'une suite d'instructions x86 (dans notre cas, un bloc de base B) dans l'environnement s.
Pour cela, nous allons commencer par traduire la première instruction x86 du bloc de base B, notée $\texttt{i}_0$, en un programme DBA $P = \ell_0 : I_0 \dots \ell_n : I_n$. De la même façon, nous traduisons l'état machine x86 s dans l'état machine DBA équivalent (ℓ, δ, γ). Ensuite, grâce à la méthode donnée ci-dessus, nous exécutons une par une les instructions DBA de P en partant de l'état machine (ℓ, δ, γ). Finalement, après exécution du programme DBA P, nous obtenons l'état machine DBA $(\ell', \delta', \gamma')$ correspondant à l'état machine obtenu après l'exécution de l'instruction x86 $\texttt{i}_0$.
En répétant cette même opération pour les instructions suivantes du bloc de base B, nous arrivons à calculer l'état machine de sortie du bloc de base : s'.
Nous notons finalement que la méthode que nous utilisons afin d'effectuer de façon effective l'exécution symbolique d'une bloc de base consiste à passer successivement du monde x86 au monde DBA et inversement. De plus, nous remarquons que les calculs se font réellement au niveau des instructions DBA.

4.5.1 Permission d'exécution des instructions x86

Il important de noter qu'une vérification supplémentaire est effectuée durant l'exécution symbolique d'un bloc de base. En effet, nous devons vérifier que chaque instruction x86 a le « droit » d'être exécutée, sinon une exception est déclenchée (voir chapitre 10).
Pour cela, avant de traduire une instruction x86 `i` en son programme DBA équivalent en vu de l'exécuter, nous vérifions (grâce à p) que chaque case mémoire sur lesquelles vivent l'opcode de `i` disposent bien des permissions en lecture et en exécution. Si ce n'est pas le cas, alors l'instruction `i` n'est pas exécutée, une exception est déclenchée et BOA calcule le nouvel état machine.

4.5.2 Gestion des exceptions

Durant l'exécution réelle d'une suite d'instructions, si une instruction déclenche une exception alors le système d'exploitation va prendre le relais afin de chercher un gestionnaire d'exception. Dans ce cas, il est possible que les instructions présentes en aval de l'instruction ayant déclenchée l'exception ne soient jamais exécutées ou bien qu'elles ne soient exécutées que plus tard.

Dans notre cas nous détectons les exceptions au niveau DBA. Afin de savoir si l'exécution d'une instruction x86 a donné lieu au déclenchement d'une exception, il nous suffit de vérifier après l'exécution de chaque instruction x86 que la valeur du pointeur d'instruction dans l'état machine correspond bien à l'adresse de l'instruction suivante dans le bloc de base. Si ce n'est pas le cas alors cela signifie qu'une exception a eu lieu. Dans ce cas les instructions restantes dans le bloc de base ne sont pas exécutées et l'état machine s' obtenu après exécution du bloc de base correspond à l'état machine obtenu après l'exécution symbolique du programme DBA correspondant à l'instruction x86 ayant déclenchée l'exception.

4.5.3 Instructions non prises en charge par le langage DBA

Certaines instructions x86 ne sont actuellement pas prises en charge par le langage DBA. C'est par exemple le cas des instructions `INT` et des instructions préfixées par un `REP`. Dans ces cas, BOA se charge d'effectuer l'exécution symbolique de ces instructions sans utiliser le langage DBA.

Chapitre 5

Fonctionnement général de BOA

BOA, pour *Basic blOck Analysis*, est une plateforme permettant d'extraire le graphe de flot de contrôle d'un binaire exécutable.

Le fonctionnement de BOA est basé sur une approche hybride combinant un algorithme de désassemblage récursif à de l'exécution symbolique, le tout à l'échelle du bloc de base. L'utilisation de l'exécution symbolique dans la plateforme nous permet d'obtenir des états partiels de la machine (valeurs des registres et des cases mémoires) en sortie des blocs de base. Ces informations nous permettent quand cela est possible de résoudre les adresses de saut des instructions de saut dynamique (`RET`, `CALL`, `JMP`, ...) rencontrées durant le désassemblage mais aussi de détecter les prédicats opaques.

BOA a été pensé pour résister aux différentes techniques d'anti-analyses utilisées par les binaires pour échapper au désassemblage et à la rétro-ingénierie. En particulier, BOA intègre des fonctionnalités génériques afin de détecter et traiter correctement les falsifications de pile d'appels (*call stack tampering*), les prédicats opaques, les auto-modifications ou encore la construction à la volée de la table d'importation.

Dans ce chapitre, nous expliquons concrètement comment la plateforme BOA fonctionne, c'est-à-dire, comment BOA, à partir d'un fichier binaire, est capable de construire le graphe de flot de contrôle correspondant. Pour cela, nous commençons par définir formellement ce que nous entendons par un graphe de flot de contrôle. Enfin, nous décrivons le fonctionnement de l'algorithme principal de BOA et plus précisément la boucle permettant la construction étape par étape du binaire analysé. Concernant les autres fonctionnalités de BOA (détection des falsifications de la pile d'appels, prédicats opaques, ...), elles sont présentées avec plus de détails dans les chapitres 6, 7, 8, 9 et 10.

5.1 Description de l'algorithme utilisé par BOA

De façon générale, la plateforme BOA prend pour entrée le fichier binaire pour lequel le graphe de flot de contrôle doit être construit. L'exécution de BOA se résume en une phase d'initialisation durant laquelle le binaire est *parsé* afin d'obtenir les informations nécessaires à son désassemblage puis d'une boucle de désassemblage durant laquelle chaque bloc de base du binaire est désassemblé afin de permettre la construction, au fur et à mesure, du graphe de flot de contrôle. La boucle de désassemblage se termine quand le graphe de flot de contrôle n'évolue plus. Finalement, différents fichiers sont générés afin de permettre l'exploitation des résultats (liste des instructions, liste des blocs de base, graphe de flot de contrôle, rapport de désassemblage contenant des informations sur les différentes obfuscations détectées, ...).

5.1.1 Fonctionnement général de la boucle principale de BOA

L'algorithme principal de BOA prend en entrée un état machine initial noté $s_0 = (\texttt{ep}, \sigma_0, p_0)$ où `ep` est le point d'entrée du binaire à analyser, σ_0 est la configuration initiale des valeurs des registres et cases mémoire et p_0 est la configuration initiale des permissions des cases mémoire.

À partir de l'état machine initial s_0, BOA construit de façon incrémentale le graphe de flot de contrôle étendu $G = (\mathbf{B}, \mathbf{E}, \texttt{ep})$. La racine de G correspond au point d'entrée de l'état machine initial.

Durant la totalité de son analyse, BOA maintient une pile de configurations, notée $\mathcal{A}$, dont les éléments sont des 5-uplets contenant un bloc de base B qui est un nœud de $\mathbf{B}$, un état machine $(\texttt{ip}, \sigma, p)$ et une pile d'adresses *return-site* π.

Étape 1 : initialisation Au démarrage de l'algorithme de BOA, le graphe de flot de contrôle est uniquement composé d'un nœud factice noté r. Nous empilons dans la pile $\mathcal{A}$ (initialement vide) la configuration initiale $(r, \texttt{ep}, \sigma_0, p_0, \epsilon)$ (points ① et ② du listing 2).

Étape 2 : boucle Après cette étape d'initialisation, BOA entre dans sa boucle principale. À chaque itération de la boucle, BOA retire de la pile $\mathcal{A}$ une configuration $(B, \texttt{ip}, \sigma, p, \pi)$ (point ③ du listing 2). Ensuite, nous résumons un tour de boucle en trois phases :

1. **Mise à jour de G :** Si $B_{\texttt{ip}}$, le bloc de base d'adresse ip dans σ, n'existe pas dans G alors BOA tente de le désassembler. Le cas échéant, la *stack closure* ρ de l'arc $(B, B_{\texttt{ip}})$ est mise à jour (point ④ du listing 2).
2. **Traitement des boucles :** Lorsque nous détectons une boucle, si le compteur de boucle n'a pas atteint *max-loop* alors nous mettons à jour ρ avec (σ, p, π'). Sinon nous commençons le calcul d'un point fixe (point ⑤ du listing 2).
3. **Calcul des adresses successeurs :** En exécutant symboliquement le bloc de base $B_{\texttt{ip}}$ nous calculons, si possible, le ou les adresses successeurs de $B_{\texttt{ip}}$ et nous empilons en conséquence les nouvelles configurations dans $\mathcal{A}$ (points ⑥ et ⑦ du listing 2).

Finalement, l'algorithme se termine lorsque la pile $\mathcal{A}$ est vide.

Nous décrivons maintenant avec plus de détails le fonctionnement de l'algorithme principal de BOA.

5.1.2 Étape 1 : initialisation de la pile $\mathcal{A}$

L'étape d'initialisation est effectuée une seule et unique fois lors de l'exécution de BOA. Elle permet de préparer BOA avant de démarrer la boucle permettant la construction du graphe de flot de contrôle. Cette étape nous permet de préparer la pile $\mathcal{A}$ avec l'état initial de la machine.

5.1.2.1 Étape 1.1 : création de l'état machine initial s_0

Nous créons un état machine s_0 avec pour environnement σ_0, pour permissions p_0 et pour adresse ep nous permettant de représenter l'état dans lequel serait la machine réelle après avoir chargé le programme en mémoire (sous-entendu, juste avant l'exécution de la première instruction du programme). Ce mécanisme est décrit dans la section 3.1.5.

Nous savons que sur une machine réelle, la première instruction du programme exécutée par le processeur est toujours l'instruction présente à l'adresse du point d'entrée du binaire. Nous initialisons alors ep par la valeur du point d'entrée que nous avons récupéré dans l'en-tête du binaire.

Concernant les fonctions σ_m, σ_r, σ_f et p, elles sont initialisées avec les valeurs que l'on rencontre lors d'un chargement réel du binaire par le système d'exploitation.

5.1.2.2 Étape 1.2 : ajout du 5-uplet initial dans $\mathcal{A}$

Avant de commencer les itérations de la boucle principale, nous créons un graphe noté G, le graphe de flot de contrôle étendu du binaire à analyser que nous allons construire au fur et à mesure des itérations de la boucle en y ajoutant un par un les blocs de base désassemblés. Nous ajoutons dans G un premier nœud factice noté r.

De plus, avant de commencer les itérations de la boucle principale, nous initialisons la pile $\mathcal{A}$ en y insérant la configuration initiale $(r, \texttt{ep}, \sigma_0, p_0, \epsilon)$ contenant le nœud factice du GFC, l'état initial de la machine calculé précédemment ainsi que la pile d'adresses *return-site* vide.

Algorithme 2 : Boucle principale de BOA

```
Function BuildCFG(binary) :
    Input : Le fichier binary à traiter
    Output : La graphe de flot de contrôle G du fichier binary
    /* ① Création de l'état machine initial */
    (r, ep, σ_0, p_0, ϵ) = BuildInitMachineState(binary)
    /* ② Initialisation de la pile A */
    A.push(r, ep, σ_0, p_0, ϵ)
    /* Boucle principale de BOA */
    while A do
        /* ③ Récupération de la configuration à traiter pour ce tour */
        (B, ip, σ, p, π) = A.pop()
        /* ④ Désassemblage du bloc de base d'adresse ip et mise à jour de G */
        (B_ip, G) = DisasBasicBlock(ip, σ,p,B,G)
        /* Calcul de la nouvelle pile de return-site */
        π' = ComputeReturnSiteStack(π, B, B_ip, G)
        /* ⑤ Gestion des boucles */
        (continue, σ, p, G) = LoopHandler(B_ip, B, π', σ, p, G)
        if continue then
            /* ⑥ Exécution symbolique de B dans σ et p */
            (ip', σ', p') = MachineState(B_ip, σ, p)
            /* ⑦ Ajout des nouvelles configurations dans A */
            if ip' ≠ ⊥ then
                A.push(B_ip, ip', σ', p', π')
            else if type(B) == COND then
                A.push(B_ip, next-mem, σ', p', π')
                A.push(B_ip, target, σ', p', π')
```

5.1.3 Étape 2 : boucle de construction du GFC

5.1.3.1 Étape 2.1 : mise à jour de G

Récupération d'une configuration dans $\mathcal{A}$ Dans la boucle, la première chose à faire est de récupérer en haut de la pile $\mathcal{A}$ une configuration composée d'un bloc de base B, d'une adresse `ip`, d'un contexte machine σ, des permissions p et d'une pile de *return-site* π :

$$(B, \texttt{ip}, \sigma, p, \pi) = \mathcal{A}.\texttt{pop()}$$

Désassemblage du bloc de base $B_{\texttt{ip}}$ Pour ce tour de boucle, nous allons travailler avec $B_{\texttt{ip}}$, le bloc de base ayant pour point d'entrée l'instruction d'adresse `ip`. La fonction `DisasBasicBlock`, donnée dans l'algorithme 4 décrit la façon dont nous obtenons ce bloc de base. Pour résumer, cet algorithme doit traiter trois cas différents :

1. $B_{\texttt{ip}} \in G$ **:** Si le bloc de base $B_{\texttt{ip}}$ est déjà un nœud du graphe G, alors nous le récupérons directement dans G.
2. $B_{\texttt{ip}} \notin G$ **mais** $\texttt{ip} \in B_{\texttt{x}}$ **et** $B_{\texttt{x}} \in G$ **:** Si $B_{\texttt{ip}}$ n'est pas un bloc de base de G, mais que l'instruction `i` d'adresse `ip` appartient à un bloc de base $B_{\texttt{x}}$ déjà présent dans G, alors nous devons couper $B_{\texttt{x}}$ en deux blocs de base différents. La procédure `SplitBasicBlock` (algorithme 5) coupe le bloc de base $B_{\texttt{x}}$ au niveau de l'instruction d'adresse `ip` afin d'obtenir, à partir du bloc de base original $B_{\texttt{x}}$, deux blocs de base (le bloc de base amputé $B_{\texttt{x}}$ et le nouveau bloc de base $B_{\texttt{ip}}$).
3. $B_{\texttt{ip}} \notin G$ **:** Si dans G aucun bloc de base ne contient l'instruction `i` d'adresse `ip`, alors nous désassemblons le bloc de base sémantique maximal $B_{\texttt{ip}}$ à partir de l'adresse `ip`. Il est possible que cette procédure ne retourne aucun bloc de base, c'est typiquement le cas si aucune instruction valide n'est présente à l'adresse `ip`. Dans ce cas, BOA agit comme une continuation et va simuler le comportement du système d'exploitation ; il va déclencher une exception de type *illegal instruction*, rechercher si un gestionnaire d'exception est présent pour traiter cette erreur et sauvegarder le contexte actuel sur la pile. Finalement, si un gestionnaire d'exception est trouvé, BOA retourne un nouvel état machine $(\texttt{ip}', \sigma', p')$ représentant la machine prête à exécuter le gestionnaire d'exception. Nous empilons alors la configuration $(B, \texttt{ip}', \sigma', p', \pi)$ sur la pile $\mathcal{A}$ et ce tour de boucle est avorté afin de revenir au début de l'étape 2. Les détails de ce cas particulier sont donnés dans la section 10.

Nous pouvons maintenant travailler avec le bloc de base $B_{\texttt{ip}}$.

Calcul de la nouvelle pile de *return-site* La mise à jour du graphe de flot de contrôle étendu G permet également la mise à jour de M_G, le système à pile auquel il est rattaché. Nous utilisons alors le système à pile M_G afin de calculer π' grâce à la transition $(B, \pi) \to (B_{\texttt{ip}}, \pi')$.

5.1.3.2 Étape 2.2 : gestion des boucles

Pour rappel, la détection des boucles dans la graphe de flot de contrôle G est assurée par le système à pile M_G. Afin de gérer les boucles durant l'analyse, nous fixons un nombre appelé *max-loop* qui va permettre de fixer le nombre de fois que BOA pourra traiter une boucle sans approximation.

Pour cela, sur l'arc $(B, B_{\texttt{ip}})$, nous mettons à jour l'entrée de la *stack closure* correspondant à la pile π'. Si le compteur n'a pas atteint *max-loop*, il est alors incrémenté et nous mettons à jour cette entrée avec l'état machine courant. Sinon nous commençons un calcul de point fixe en réduisant l'état machine.

Cas n°1 : compteur $<$ *max-loop* Étant donné ρ, la *stack closure* de l'arc $(B, B_{\texttt{ip}})$, si nous sommes dans le cas où l'arc $(B, B_{\texttt{ip}})$ n'a jamais été rencontré avec la pile π', autrement dit si $\pi' \notin dom(\rho)$ alors le domaine de ρ est étendu par la pile de *return-site* π', c'est-à-dire $dom(\rho) = dom(\rho) \cup \{\pi'\}$. De plus, nous associons dans ρ l'entrée π' au triplet composé de l'environnement machine σ, des permissions mémoire p et d'un compteur initialisé à un :

$$\rho[\pi'] = (\sigma, p, 1)$$

Sinon si *cnt*, le compteur présent sur cet arc pour la pile π' est inférieur à *max-loop* alors nous augmentons la valeur du compteur de 1 et nous mettons à jour cette entrée avec le contexte et les permissions machine actuels :

$$\rho[\pi'] = (\sigma, p, cnt + 1)$$

Cas n°2 : compteur $\geq$ *max-loop* Étant donné ρ, la *stack closure* de l'arc $(B, B_{\texttt{ip}})$ et *cnt* le compteur présent dans le triplet $\rho[\pi']$. Si $cnt \geq$ *max-loop*, ou autrement dit si l'arc $(B, B_{\texttt{ip}})$ a déjà été rencontré plus de *max-loop* fois pour la pile π' alors nous faisons le nécessaire afin de savoir si nous continuons l'exploration de ce chemin d'exécution ou non.

Nous commençons par récupérer l'environnement machine $(\sigma_{prev}, p_{prev})$ dans $\rho[\pi']$.

Ensuite, nous calculons un environnement machine réduit noté $(\sigma_{merge}, p_{merge})$ à partir de la combinaison de l'environnement courant (σ, p) et de l'environnement $(\sigma_{prev}, p_{prev})$:

$$(\sigma_{merge}, p_{merge}) = \texttt{MergeMachineContext}\big((\sigma, p), (\sigma_{prev}, p_{prev})\big)$$

Enfin, nous mettons à jour l'entrée associée à π' dans ρ en y plaçant l'environnement machine réduit $(\sigma_{merge}, p_{merge})$ que nous venons de calculer :

$$\rho[\pi'] = (\sigma_{merge}, p_{merge}, cnt + 1)$$

Si $(\sigma_{merge}, p_{merge})$ et $(\sigma_{prev}, p_{prev})$ sont identiques, cela signifie que nous avons atteint un point fixe à force de réduire cet environnement machine. Il est donc inutile de continuer à explorer ce chemin d'exécution. Dans ce cas, aucune nouvelle configuration n'est empilée dans $\mathcal{A}$ et si cette dernière n'est pas vide, un nouveau tour de boucle démarre à l'étape 2.1.

Dans le cas contraire, nous continuons l'exploration. Cependant, pour l'exécution symbolique du bloc de base $B_{\texttt{ip}}$ (étape 2.3), nous n'utilisons pas l'environnement machine courant (σ, p) mais l'environnement réduit $(\sigma_{merge}, p_{merge})$ que nous venons de calculer. Enfin, nous passons à l'étape 2.3.

Conclusion Ce mécanisme permet petit à petit de faire tendre l'ensemble des variants d'une boucle vers la valeur $\bot$ (car l'opération de fusion `MergeMachineContext` fait perdre de l'information). Une fois que tous les variants de la boucle actuelle sont à $\bot$ alors le contexte machine n'évolue plus, il n'est donc pas nécessaire de continuer l'exploration.

Remarques d'implémentation

1. Nous considérons que seulement les instructions de type saut conditionnel ou saut dynamique sont susceptibles d'engendrer une boucle dans un graphe de flot de contrôle. Partant de ce postulat, nous décidons d'étiqueter par la fonction ρ seulement les arcs ayant comme source ce type d'instructions. Pour les autres, l'étape 2.2 n'est pas exécutée.
2. Nous pouvons remarquer que pour un arc et pour une pile de *return-site* donnés, il est nécessaire de sauvegarder l'environnement machine s seulement une fois que la valeur du compteur a atteint *max-loop* $- 1$.

5.1.3.3 Étape 2.3 : recherche des adresses successeurs

Durant cette étape nous construisons l'ensemble A_{succ} contenant les adresses candidates comme successeurs pour le bloc de base $B_{\texttt{ip}}$. Nous commençons par calculer l'état de la machine après l'exécution de $B_{\texttt{ip}}$: $(\texttt{ip}, \sigma, p) \xrightarrow{B_{\texttt{ip}}} (\texttt{ip}', \sigma', p')$. Pour cela, nous utilisons la procédure détaillée dans le chapitre 4.

Comme cela a été expliqué dans la section 4.5.2, durant l'exécution symbolique du bloc de base, il est possible qu'une instruction déclenche une exception, conduisant alors une partie des instructions de $B_{\texttt{ip}}$ à ne pas être exécutée. Dans ce scénario, deux cas sont à prendre en compte. Si c'est la première fois que $B_{\texttt{ip}}$ est traitée par BOA alors les instructions de $B_{\texttt{ip}}$ non exécutées sont simplement retirées du bloc de base, nous considérons alors l'instruction ayant déclenchée l'exception comme la dernière instruction de $B_{\texttt{ip}}$. Dans le cas contraire, si le bloc de base $B_{\texttt{ip}}$ a précédemment été traité par BOA, alors cela signifie

que les instructions présentes en aval de l'instruction ayant causée l'exception sont « vivantes ». Dans ce cas le bloc de base $B_{\texttt{ip}}$ est coupé en deux afin d'isoler les instructions ici non exécutées du reste du bloc de base.

Dans tous les cas, nous avons maintenant $(\texttt{ip}', \sigma', p')$, l'état machine obtenu après exécution du bloc de base $B_{\texttt{ip}}$ dans l'état machine $(\texttt{ip}, \sigma, p)$.

Nous considérons maintenant deux cas, le premier quand $\texttt{ip}' \neq \bot$ et le second quand $\texttt{ip}' = \bot$.

Cas $\texttt{ip}' \neq \bot$ Ce cas est pour nous le plus simple, nous avons simplement $A_{succ} = \{\texttt{ip}'\}$.

Cas $\texttt{ip}' = \bot$ Dans ce cas, nous construisons l'ensemble A_{succ} en fonction du type de la dernière instruction de $B_{\texttt{ip}}$ notée $\texttt{i}_n$:

- **Si $\texttt{i}_n$ est un saut conditionnel,** alors nous considérons que les deux branches du `Jcc` sont vivantes. Nous notons $\texttt{a}_{next\text{-}mem}$ l'adresse suivante en mémoire et $\texttt{a}_{target}$ l'adresse de saut donnée par l'instruction. Nous avons alors $A_{succ} = \{\texttt{a}_{next\text{-}mem}, \texttt{a}_{target}\}$.
- **Sinon,** dans tous les autres cas nous considérons ne pas être en mesure de connaître la ou les adresses candidates à $B_{\texttt{ip}}$ lors d'une exécution à partir de l'état machine $(\texttt{ip}, \sigma, p)$. Nous avons alors $A_{succ} = \emptyset$.

À cet instant, l'ensemble A_{succ} contient 0, 1 ou 2 adresses. Pour chaque adresse $\texttt{a}_i \in A_{succ}$, nous empilons dans $\mathcal{A}$ la configuration $(B_{\texttt{ip}}, \texttt{a}_i, \sigma', p', \pi')$. Finalement, si à cet instant la pile $\mathcal{A}$ n'est pas vide, nous revenons au début de la boucle, à l'étape 2.1, afin de traiter une nouvelle configuration de $\mathcal{A}$.

Algorithme 3 : Désassemblage d'une instruction

```
Function DisasInstr(ip, σ, p) :
    Input : L'adresse mémoire de l'instruction à désassembler (ip)
            Le contexte mémoire (σ)
            Les permissions mémoire (p)
    Output : L'instruction valide à l'adresse ip ou null sinon
    opcode = [ ]
    for j ∈ [0, 15] do
        perm = p[ip + j]
        if perm ∉ {RX, RWX} then
            return null
        byte = σ[ip + j]
        if byte == ⊥ then
            return null
        opcode = opcode + byte
        if IsValidInstr(opcode) then
            return Instr(opcode)
    return null
```

Algorithme 4 : Désassemblage d'un bloc de base et mise à jour du graphe de flot de contrôle

Function `DisasBasicBlock(`ip`,` σ`,` p`,` B`,` G`)` **:**
- **Input :** L'adresse mémoire du bloc de base à désassembler (`ip`)
  Le contexte mémoire (σ)
  Les permissions mémoire (p)
  Le bloc de base antécédent (B)
  Le graphe de flot de contrôle (G)
- **Output :** Le bloc de base à l'adresse `ip`
  Le graphe de flot de contrôle G mis à jour
- $B_{\texttt{ip}} = [\,]$
- `/* Si le bloc de base d'adresse ip existe déjà dans` G `*/`
- **if** $B_{ip} \in G_{\mathbf{B}}$ **then**
  - $B_{\texttt{ip}} = G_{\mathbf{B}}[B_{\texttt{ip}}]$
- `/* Si un bloc de base dans` G `contient déjà une instruction d'adresse ip */`
- **else if** $\exists B_x \in G_{\mathbf{B}}$ *tel que* $ip \in B_x$ **then**
  - $(B_{\texttt{ip}}, G) =$ `SplitBasicBlock(`ip`,` G`)`
- `/* Sinon on désassemble le bloc de base maximal d'adresse ip */`
- **else**
  - **while** *true* **do**
    - **if** $\exists B_x \in G_{\mathbf{B}}$ *tel que* $ip \in B_x$ **then**
      - break
    - $i =$ `DisasInstr(`ip`,` σ`,` p`)`
    - **if** $i ==$ `null` **then**
      - break
    - $B_{\texttt{ip}} = B_{\texttt{ip}} + i$
    - **if** $type(i) \neq SEQ$ **then**
      - break
    - `ip` $=$ `ip` $+ size(i)$
- **if** $B_{ip} \neq [\,]$ **then**
  - `/* Si besoin on ajoute l'arc entre` B `et` $B_{\texttt{ip}}$ `*/`
  - $G_{\mathbf{E}} = G_{\mathbf{E}} + (B, B_{\texttt{ip}})$
- **return** $(B_{\texttt{ip}}, G)$

Algorithme 5 : Division d'un bloc de base en deux

Function `SplitBasicBlock(`a, G`)` **:**
- **Input :** Le point d'entrée du nouveau bloc de base (a)
  Le graphe de flot de contrôle (G)
- **Output :** Le nouveau bloc de base ($B_{\mathtt{a}}$)
  Le graphe de flot de contrôle G mis à jour
- `/* Recherche du bloc de base` $B_{\mathtt{x}}$ `contenant l'adresse` a `*/`
- *let* $B_{\mathtt{x}} \in G_{\mathbf{B}}$ tel que $a \in B_{\mathtt{x}}$
- `/* Déplacement d'une partie des instructions de` $B_{\mathtt{x}}$ `dans` $B_{\mathtt{a}}$ `*/`
- $move = false$
- $B_{\mathtt{a}} = [\,]$
- **foreach** $(\alpha, i_\alpha) \in B_x$ **do**
  - **if** $a == \alpha$ **then**
    - $move = true$
  - **if** $move$ **then**
    - $B_{\mathtt{a}} = B_{\mathtt{a}} + (\alpha, i_\alpha)$
    - $B_{\mathtt{x}}.remove(\alpha)$
- `/* Ajout de` $B_{\mathtt{a}}$ `dans` G `*/`
- $G_{\mathbf{B}} = G_{\mathbf{B}} + B_{\mathtt{a}}$
- `/* Ajout de l'arc entre` $B_{\mathtt{x}}$ `et` $B_{\mathtt{a}}$ `*/`
- $G_{\mathbf{E}} = G_{\mathbf{E}} + (B_{\mathtt{x}}, B_{\mathtt{a}})$
- **return** (B_a, G)

Algorithme 6 : Fusion de deux contextes machine

Function `MergeVal`(v_1, v_2) **:**
- **Input :** v_1 et v_2, les deux valeurs à fusionner
- **Output :** v, le résultat de la fusion
- **if** $v_1 == \bot \vee v_2 == \bot$ **then**
  - **return** $\bot$
- **else if** $v_1 \neq v_2$ **then**
  - **return** $\bot$
- **else**
  - **return** v_1

Function `MergePerms`(p_1, p_2) **:**
- **Input :** p_1 et p_2, les deux ensembles de permission à fusionner
- **Output :** p, le résultat de la fusion
- **return** $p_1 \cap p_2$

Function `MergeMachineContext`((σ_1, p_1), (σ_2, p_2)) **:**
- **Input :** (σ_1, p_1) et (σ_2, p_2), les deux environnements machine à fusionner
- **Output :** (σ, p), le résultat de la fusion
- **forall** $r \in \mathbb{R}eg$ **do**
  - $\sigma_r[r] =$ `MergeVal`($\sigma_{1_r}[r]$, $\sigma_{2_r}[r]$)
- **forall** $f \in \mathbb{F}lag$ **do**
  - $\sigma_f[f] =$ `MergeVal`($\sigma_{1_f}[f]$, $\sigma_{2_f}[f]$)
- **forall** $a \in \mathbb{A}ddr$ **do**
  - $\sigma_m[a] =$ `MergeVal`($\sigma_{1_m}[a]$, $\sigma_{2_m}[a]$)
  - $p[a] =$ `MergePerms`($p_1[a]$, $p_2[a]$)
- **return** σ

Algorithme 7 : Gestion des boucles

Function `LoopHandler(`B_{ip}, B, π, σ, p, G`)` **:**
- **Input :** Le bloc de base courant ($B_{\texttt{ip}}$)
  Le bloc de base antécédent (B)
  La pile de *return-site* courante (π)
  Le contexte mémoire courant (σ)
  Les permissions mémoire courantes (p)
  La graphe de flot de contrôle (G)
- **Output :** Un booléen indiquant si l'analyse de ce chemin d'exécution doit continuer
  Le contexte et les permissions machine à utiliser
  Le graphe de flot de contrôle mis à jour
- `/* Récupération du compteur sur l'arc` $(B, B_{\texttt{ip}})$ `pour` π `et incrémentation */`
- **if** $\pi \notin dom(\rho)$ **then**
  - $cnt = 1$
- **else**
  - $(\sigma_{prev}, p_{prev}, cnt) = \rho[\pi]$
  - $cnt = cnt + 1$
- `/* Cas n°1 : compteur <` *max-loop* `*/`
- **if** $cnt \leq$ max-loop **then**
  - $\rho[\pi] = (cnt, \sigma, p)$
  - **return** $(true, \sigma, p, G)$
- `/* Cas n°2 : compteur ≥` *max-loop* `*/`
- **else**
  - $(\sigma_{merge}, p_{merge})$ = `MergeMachineContext(`(σ, p), $(\sigma_{prev}, p_{prev})$`)`
  - $\rho[\pi] = (cnt + 1, \sigma_{merge}, p_{merge})$
  - **if** $(\sigma_{merge}, p_{merge}) == (\sigma_{prev}, p_{prev})$ **then**
    - **return** $(false, \sigma_{merge}, p_{merge}, G)$
  - **return** $(true, \sigma_{merge}, p_{merge}, G)$

5.2 Exemples d'exécutions de BOA

5.2.1 Détection d'une exception

Afin de montrer comment BOA gère la présence d'une exception dans un binaire, nous avons compilé le programme assembleur donné dans le listing 5.1 et nous l'avons fait analyser par BOA. Pour des raisons de lisibilité, le programme analysé dans cet exemple n'a aucun but ou sens précis, il contient un nombre restreint d'instructions dans un unique but de démonstration.

Listing 5.1 – Programme déclenchant une exception

```
main:
    mov eax, 0x6 ; eax <-- 0x6
    mov ebx, 0x1 ; ebx <-- 0x1
    jmp A        ; goto A

A:
    xor edx, edx ; edx <-- 0x0
    div ebx      ; edx:eax <-- edx:eax / ebx
    dec ebx      ; ebx <-- ebx - 0x1
    cmp eax, 0x6 ; r <-- eax - 0x6
    jz A         ; goto A if r == 0x0
```

Le déroulement simplifié de l'algorithme de BOA avec cet exemple est donné dans le table 5.1.

5.2.2 Gestion d'une boucle

Le programme assembleur donné dans le listing 5.2 ne fait rien d'intéressant si ce n'est d'ajouter 1 à la valeur du compteur `EAX` dans une boucle et ce, jusqu'à ce que celui-ci atteigne la valeur `0x1000`. Nous souhaitons montrer avec cet exemple simple le fonctionnement de calcul de point fixe effectué par BOA lorsqu'il détecte qu'une boucle est exécutée plus de *max-loop* fois. Comme ce programme ne contient aucune fonction, nous émettons ici l'utilisation des piles de *return-site* et nous stockons les compteurs de boucle directement sur les arcs.

Listing 5.2 – Programme simple avec une boucle

```
main:
    xor eax, eax     ; eax <-- 0x0
    jmp A            ; goto A

A:
    inc eax          ; eax <-- eax + 0x1
    cmp eax, 0x1000 ; r <-- eax - 0x1000
    jnz A            ; goto A if r != 0x0

    ret
```

Le déroulement simplifié de l'algorithme de BOA avec cet exemple est donné dans le table 5.2. Nous prenons dans notre cas un *max-loop* de 2.

TABLE 5.1 – Exemple d'analyse d'un programme déclenchant une exception

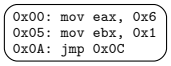
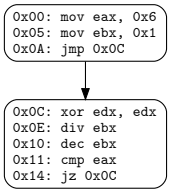
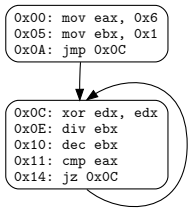
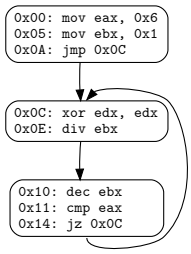
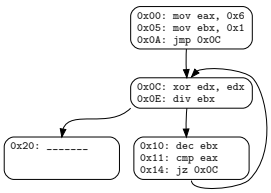
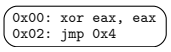
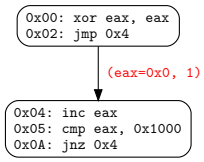
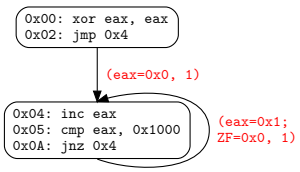
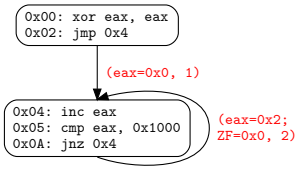
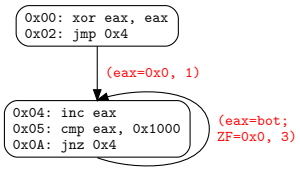
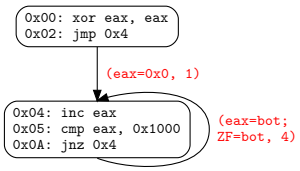
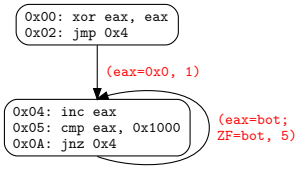
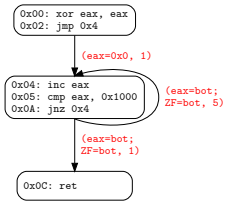
	Configuration traitée	**GFC**	**Configurations empilées**	**Remarques**
Tour n°1	$(r, \texttt{0x00}, \sigma_0)$		$(B_{\texttt{0x00}}, \texttt{0x0C}, \sigma_1)$	$\sigma_1 = \begin{cases} \texttt{EAX} = \texttt{0x6} \\ \texttt{EBX} = \texttt{0x1} \end{cases}$
Tour n°2	$(B_{\texttt{0x00}}, \texttt{0x0C}, \sigma_1)$		$(B_{\texttt{0x0C}}, \texttt{0x0C}, \sigma_2)$	$\sigma_2 = \begin{cases} \texttt{EAX} = \texttt{0x6} \\ \texttt{EBX} = \texttt{0x0} \\ \texttt{EDX} = \texttt{0x0} \\ \texttt{ZF} = \texttt{0x1} \end{cases}$
Tour n°3	$(B_{\texttt{0x0C}}, \texttt{0x0C}, \sigma_2)$		(voir tour n°3 bis)	L'instruction `0x0E: div ebx` déclenche une exception due à une division par zéro. Le bloc de base `0x0C` doit alors être divisé en deux.
Tour n°3 bis	(voir tour n°3)		$(B_{\texttt{0x0C}}, \texttt{0x20}, \sigma_3)$	Nous faisons ici l'hypothèse que le gestionnaire d'exception calculé par BOA se situe à l'adresse `0x20`.
Tour n°4	$(B_{\texttt{0x0C}}, \texttt{0x20}, \sigma_3)$		[...]	Nous arrêtons ici le déroulement de cet exemple bien que lors d'une analyse réelle BOA continuerait son exécution jusqu'à ce que la pile $\mathcal{A}$ soit vide.

TABLE 5.2 – Exemple d'analyse d'un programme contenant une boucle

	Configuration traitée	GFC	Configurations empilées	Remarques
Tour n°1	$(r, \texttt{0x00}, \sigma_0)$		$(B_{\texttt{0x00}}, \texttt{0x04}, \sigma_1)$	$\sigma_1 = \begin{cases} \texttt{EAX} = \texttt{0x0} \end{cases}$
Tour n°2	$(B_{\texttt{0x00}}, \texttt{0x04}, \sigma_1)$		$(B_{\texttt{0x04}}, \texttt{0x04}, \sigma_2)$	L'arc $(B_{\texttt{0x00}}, B_{\texttt{0x04}})$ n'a jamais été rencontré. $\sigma_2 = \begin{cases} \texttt{EAX} = \texttt{0x1} \\ \texttt{ZF} = \texttt{0x0} \end{cases}$
Tour n°3	$(B_{\texttt{0x04}}, \texttt{0x04}, \sigma_2)$		$(B_{\texttt{0x04}}, \texttt{0x04}, \sigma_3)$	L'arc $(B_{\texttt{0x04}}, B_{\texttt{0x04}})$ n'a jamais été rencontré. $\sigma_3 = \begin{cases} \texttt{EAX} = \texttt{0x2} \\ \texttt{ZF} = \texttt{0x0} \end{cases}$
Tour n°4	$(B_{\texttt{0x04}}, \texttt{0x04}, \sigma_3)$		$(B_{\texttt{0x04}}, \texttt{0x04}, \sigma_4)$	cnt sur l'arc $(B_{\texttt{0x04}}, B_{\texttt{0x04}})$ est égal à 1 donc $cnt < max\text{-}loop$. On incrémente le compteur. $\sigma_4 = \begin{cases} \texttt{EAX} = \texttt{0x3} \\ \texttt{ZF} = \texttt{0x0} \end{cases}$
Tour n°5	$(B_{\texttt{0x04}}, \texttt{0x04}, \sigma_4)$		$(B_{\texttt{0x04}}, \texttt{0x0C}, \sigma_5)$ $(B_{\texttt{0x04}}, \texttt{0x04}, \sigma_5)$	cnt sur l'arc $(B_{\texttt{0x04}}, B_{\texttt{0x04}})$ est égal à 2 donc $cnt \geq max\text{-}loop$. $\sigma_{merge} = \begin{cases} \texttt{EAX} = \bot \\ \texttt{ZF} = \texttt{0x0} \end{cases}$ $\sigma_5 = \begin{cases} \texttt{EAX} = \bot \\ \texttt{ZF} = \bot \end{cases}$
Tour n°6	$(B_{\texttt{0x04}}, \texttt{0x04}, \sigma_5)$		$(B_{\texttt{0x04}}, \texttt{0x04}, \sigma_6)$	cnt sur l'arc $(B_{\texttt{0x04}}, B_{\texttt{0x04}})$ est égal à 3 donc $cnt \geq max\text{-}loop$. $\sigma_{merge} = \begin{cases} \texttt{EAX} = \bot \\ \texttt{ZF} = \bot \end{cases}$ $\sigma_6 = \begin{cases} \texttt{EAX} = \bot \\ \texttt{ZF} = \bot \end{cases}$
Tour n°7	$(B_{\texttt{0x04}}, \texttt{0x04}, \sigma_6)$		$\varnothing$	cnt sur l'arc $(B_{\texttt{0x04}}, B_{\texttt{0x04}})$ est égal à 4 donc $cnt \geq max\text{-}loop$. $\sigma_{merge} = \begin{cases} \texttt{EAX} = \bot \\ \texttt{ZF} = \bot \end{cases}$ $\sigma_{merge} == \sigma_{prev}$ donc aucune configuration n'est ajoutée.
Tour n°8	$(B_{\texttt{0x04}}, \texttt{0x0C}, \sigma_5)$		[...]	[...]

Chapitre 6

Sauts dynamiques et falsifications de la pile d'appels

Dans ce chapitre nous revenons sur le choix que nous avons effectué afin de traiter le calcul des adresses de saut des instructions de type saut dynamique (instructions `RET` y compris). Nous expliquons également la méthode que nous avons implémentée afin de détecter les obfuscations de type falsification de la pile d'appels. Enfin, nous présentons les expériences que nous avons réalisées afin de valider ces fonctionnalités de BOA.

6.1 Objectifs

Comme nous l'avons vu dans l'introduction (sections 2.2.1 et 2.2.2), les instructions de saut dynamique, qu'elles soient effectuées à l'aide d'un `CALL`, d'un `JMP` ou bien encore d'un `RET`, compliquent le désassemblage et la construction du graphe de flot de contrôle des binaires.

Pour rappel, les instructions de saut dynamique sont les instructions pour lesquelles `ip` est chargé avec une valeur dépendante de l'état actuel de la machine. Ces instructions peuvent être divisées en différentes catégories en fonction des éléments de la machine qu'elles utilisent pour calculer l'adresse de saut :

1. **Registre :** Les instructions de la forme `CALL reg` ou `JMP reg` permettent de charger `ip` avec la valeur présente dans le registre spécifié par l'instruction (ex. `JMP EAX`).
2. **Mémoire :** Les instructions de la forme `CALL [addr]` ou `JMP [addr]` permettent de charger `ip` avec la valeur donnée par les quatre octets des cases mémoire *addr* à *addr* + 3 (ex. `CALL [0x12345678]`).
3. **Registre et mémoire :** Les instructions de la forme `CALL [reg]` ou `JMP [reg]` font intervenir deux variables, elles permettent de charger `ip` avec la valeur donnée par les quatre octets en mémoire dont l'adresse est donnée par la valeur du registre *reg* (ex. `JMP [EBX]`). Pour calculer l'adresse de saut d'une instruction de ce type, le défi est double puisque il est nécessaire de connaître à la fois la valeur du registre *reg* mais aussi la valeur des quatre cases mémoire pointées par *reg*.
4. **Instruction `RET` :** Cette instruction est un saut dynamique particulier car elle permet d'effectuer deux actions : la première consiste à charger `ip` avec la valeur 32 bits présente en haut de la pile et la seconde va retirer l'élément présent en haut de la pile. On peut voir l'exécution d'un `RET` comme l'exécution d'un `JMP [ESP]` suivi d'un `POP`.

Étant donné l'ensemble des instructions de saut dynamique que BOA désassemble dans un binaire, nous souhaitons être en mesure de calculer les différentes adresses de saut qu'elles peuvent atteindre, et ce, quel que soit la catégorie du saut dynamique. Concernant les `RET`, nous souhaitons pouvoir calculer leurs adresses de saut, qu'ils soient concernés par une falsification de la pile d'appels ou non.

Notre deuxième objectif est de pouvoir détecter dans un code binaire la présence de falsifications de la pile d'appels. Cet objectif n'a aucune incidence sur le désassemblage ou la construction du graphe de flot de contrôle.

6.2 État de l'art

La présence des sauts dynamiques dans les binaires n'est pas forcément due à des obfuscations. Par exemple les compilateurs utilisent habituellement ce type d'instructions pour les *switch* comme l'expliquent Dobrescu et Meignan [30] qui proposent une extension IDA afin de détecter ce type de structures.

Concernant la catégorisation de ces instructions, Shoshitaishvili et al. [68] s'orientent vers une approche plus « haut niveau » en séparant les sauts dynamiques *computed* utilisés pour les *switch*, les *context-sensitive* pour le passage d'une fonction *callback* en paramètre et enfin les *object-sensitive* utilisés dans le cadre des objets polymorphiques d'un langage orienté objet.

Finalement, peu importe la catégorie ou la nature des sauts dynamiques, ils font partie des éléments qui compliquent la construction d'un graphe de flot de contrôle comme le rappellent Meng et Miller [51]. Malgré cela, certaines applications nécessitent de disposer d'un graphe de flot de contrôle correct comme par exemple les outils de *binary rewriting* tels que *Multiverse* proposé par Bauman, Lin et Hamlen [12] ou encore *RetroWrite* développé par Dinesh et al. [27].

Une des techniques les plus utilisées afin de résoudre les adresses des instructions de sauts dynamiques (`RET` y compris) est de construire une trace d'exécution avec des outils d'instrumentation comme un *PinTool* ou *Dynamorio*. Cette technique a l'avantage que les adresses de saut calculées sont sûres car obtenues à l'issue d'une exécution réelle. Cependant cette technique possède plusieurs inconvénients. Tout d'abord une trace d'exécution ne représente qu'un seul chemin d'exécution du binaire, il est alors possible que différents chemins révèlent de nouvelles instructions de saut dynamique et/ou de nouvelles adresses cibles. Un autre problème concerne les différentes techniques d'anti-analyse utilisées par certains logiciels malveillants afin de détecter les environnements de bac à sable et ainsi arrêter prématurément leur exécution. L'instrumentation doit alors être capable de détecter ce genre de techniques afin d'appliquer des contres mesures pour espérer obtenir une trace d'exécution correcte. Cette approche dynamique est utilisée par Bonfante et al. [15] dans l'outil *CoDisasm*, par Roundy et Miller [60] dans la plateforme *SD-Dyninst* ou encore dans la méthode hybride proposée par Nguyen et al. [53].

D'un autre côté, en adoptant une solution entièrement statique, il est possible de calculer certaines adresses de sauts en utilisant des techniques à base de propagation de constantes. Nous retrouvons cette approche statique dans l'outil *Jakstab* développée par Kinder et al. [42, 43], dans la plateforme *BitBlaze* proposée par Song et al. [69] ou encore dans l'outil proposé par Bardin, Herrmann et Védrine [10].

Concernant la détection de falsifications de la pile d'appels, Qiu, Su et Ma [57] proposent une approche permettant de détecter dans une trace d'exécution les `RET` considérés comme corrompus et de les remplacer par des instructions équivalentes plus explicites utilisant des *jump*. D'un autre coté, David [25] utilise également une trace d'exécution combinée à de l'exécution symbolique arrière afin de conclure sur la falsification d'un `RET`.

6.3 Calcul des adresses de saut des sauts dynamiques

Le calcul des adresses de saut des instructions de saut dynamique (indépendamment de sa catégorie) est explicitement géré dans l'algorithme principal de BOA. En effet, comme nous l'avons expliqué dans le chapitre 5, l'algorithme de BOA est basé sur une boucle dans laquelle un bloc de base est traité à chaque tour. Pour rappel, quand BOA traite un bloc de base ayant pour dernière instruction un saut dynamique, il va utiliser une analyse de teinte afin de savoir si la valeur utilisée comme adresse de saut est concrète étant donné l'état de la machine actuel. Si le résultat s'avère être positif, BOA va alors utiliser un solveur SMT afin de calculer cette valeur et donc l'adresse de saut de l'instruction. Si BOA doit traiter à nouveau ce bloc de base, il va essayer de calculer une nouvelle adresse de saut en fonction du nouvel état machine courant et ainsi de suite à chaque fois que BOA aura re-traité ce bloc de base. Finalement, pour une instruction de saut donnée, nous obtenons l'ensemble de ses adresses de saut en regroupant les adresses obtenues à chaque passage de BOA sur le bloc de base correspondant.

L'avantage de cette méthode réside dans le fait que les adresses calculées sont correctes. En effet, si l'analyse de teinte nous indique que l'adresse de saut est concrète alors cela signifie que l'ensemble des éléments intervenant dans le calcul de cette valeur sont eux aussi connus et proviennent du code lui

même. D'un autre côté, si à la fin de son exécution BOA n'a pas été en mesure de calculer l'ensemble des chemins possibles menant à un saut dynamique, alors il est possible que toutes les adresses de saut de cette instruction n'aient pas été trouvées par BOA. Nous présentons ci-dessous plusieurs exemples pour lesquels le calcul des sauts peut être incomplet.

6.3.1 Limitations

6.3.1.1 Saut dépendant d'une entrée

Si une instruction de saut dynamique est utilisée dans le cadre d'un *switch*, et que la condition évaluée par ce *switch* est dépendante d'une entrée extérieure (par exemple un argument ou bien le résultat d'un appel à une fonction externe), alors BOA ne sera pas en mesure de calculer les adresses de saut de cette instruction. Nous pouvons également être face à ce même problème dans le cadre du polymorphisme et de l'utilisation d'une méthode virtuelle.

6.3.1.2 Approximation due aux boucles

Nous avons expliqué dans le chapitre 5 que dans le cas d'une boucle trop « longue » BOA effectue une approximation visant à sortir de la boucle en ne conservant seulement que les informations invariantes. Si ce choix nous permet de gagner du temps en avortant prématurément l'exécution d'une boucle, il a aussi pour conséquence la perte d'informations (valeurs de registres et/ou cases mémoire) dans l'état de la machine en sortie de boucle. Par conséquent, si un saut dynamique est présent en aval d'une boucle et que celui-ci utilise dans le calcul de son adresse de saut une variable perdue lors de l'approximation effectuée au niveau de la boucle alors BOA ne sera pas capable de calculer son ou ses adresses de saut.

6.3.1.3 Graphe de flot de contrôle incomplet

Il est tout à fait possible que le désassemblage effectué par BOA ne soit pas complet (par exemple dans le cas d'un saut dynamique non résolu). De ce fait, il manquera dans le graphe de flot de contrôle final certains arcs et blocs de base qui n'auront pas été désassemblés. Il est alors possible que certains chemins d'exécution n'aient pas été parcourus par BOA. Nous pouvons imaginer que certains de ces chemins mènent certains sauts dynamiques à des adresses de saut différentes que celles précédemment calculées.

6.4 Détection des falsifications de la pile d'appels

Durant l'exécution symbolique effectuée par BOA, le fait de maintenir ensemble le système à pile M_G et le graphe de flot de contrôle G auquel il est rattaché nous permet de détecter les falsifications de le pile d'appels.

En effet, l'adresse *return-site* est l'adresse empilée sur la pile par l'instruction `CALL` permettant au `RET` en aval de revenir dans la fonction appelante. Dans BOA, lorsque nous passons une instruction `CALL` dans le graphe de flot de contrôle, son adresse *return-site* est empilée dans le nouvel état de M_G. Ainsi, lorsque BOA va calculer l'adresse de saut d'un `RET` en aval, il lui suffira de comparer cette adresse à la valeur du *return-site* présent en haut de la pile. Si l'adresse de saut est identique au *return-site* présent en haut de la pile de *return-site* alors le `RET` revient bien sur l'adresse précédemment sauvegardée par le `CALL` (pas de falsification). Dans le cas contraire, cela signifie que l'adresse de retour sauvegardée par le `CALL` a été falsifiée durant l'exécution, conduisant le `RET` à sauter sur une adresse inattendue.

La figure 6.1 montre un cas sans falsification (à gauche) et un cas avec falsification de la pile d'appels (à droite).

6.4.1 Remarque sur le choix des règles de transitions dans le système à pile

Nous discutons ici des choix effectués concernant les règles qui régissent la mise à jour des états de notre système à pile.

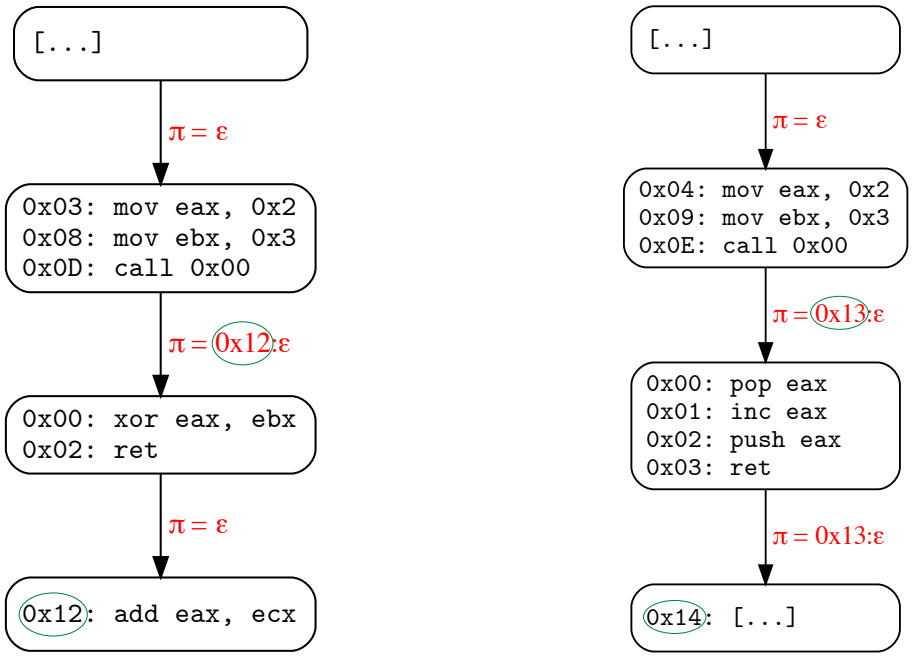


FIGURE 6.1 – Exemple de détection d'une falsification de la pile d'appels

Prenons par exemple le graphe de flot de contrôle G donné dans la figure 6.2. L'évolution de la pile de *return-site* du système à pile M_G est donnée en rouge. On remarque que l'adresse de retour utilisée par le `RET` d'adresse `0x303` a été falsifiée par la séquence d'instructions `POP-INC-PUSH`. BOA détecte bien que ce `RET` est victime d'une falsification de la pile d'appels car il saute à l'adresse `0x206` alors qu'on attend un retour en `0x205`. Le `RET` suivant, à l'adresse `0x206`, est lui aussi considéré comme falsifié. En effet, il saute à l'adresse `0x105` alors que le *return-site* présent en haut de la pile est l'adresse `0x205`. Pourtant, on aurait envie de dire que ce `RET` est bien valide car (i) il semble se situer dans la fonction ayant pour point d'entrée `0x200` et (ii) il permet de revenir en dessous du `CALL` d'adresse `0x100` qui a fait l'appel à la fonction d'adresse `0x200`.

Une solution possible afin de faire en sorte de détecter le dernier `RET` comme non falsifié serait de modifier la règle n°2 de transition du système à pile de la façon suivante : en plus d'effectuer le *pop* quand le point d'entrée du bloc de base destination correspond à l'adresse présente en haut de la pile, on effectue également le *pop* dès qu'on passe une instruction `RET`.

En appliquant cette nouvelle règle, on obtient de nouveaux états pour M_G donnés en vert dans la figure 6.2. On remarque que maintenant, le `RET` d'adresse `0x303` est bien marqué comme falsifié (saut sur l'adresse `0x206` alors qu'on attend un saut sur l'adresse `0x205`) et le `RET` d'adresse `0x206` et quant à lui maintenant marqué comme non falsifié (saut sur l'adresse `0x105`, identique à l'adresse attendue).

Cependant, en prenant un autre exemple de falsification (donné dans la figure 6.3), ici avec un simple `PUSH` on remarque que le problème s'inverse : en appliquant les nouvelles règles en vert, les deux `RET` sont détectés comme falsifiés alors qu'en prenant les règles initiales (en rouge sur la figure) le `RET` d'adresse `0x206` est bien détecté comme authentique.

Finalement, nous remarquons que dans un chemin d'exécution, si une falsification de la pile est effectuée alors la détection des `RET` en aval de cette falsification devient difficile. Nous conservons cependant les règles énoncées dans la définition de notre système à pile en gardant à l'esprit qu'il est délicat de statuer sur l'état des instructions `RET` d'un chemin d'exécution en aval d'une falsification de la pile.

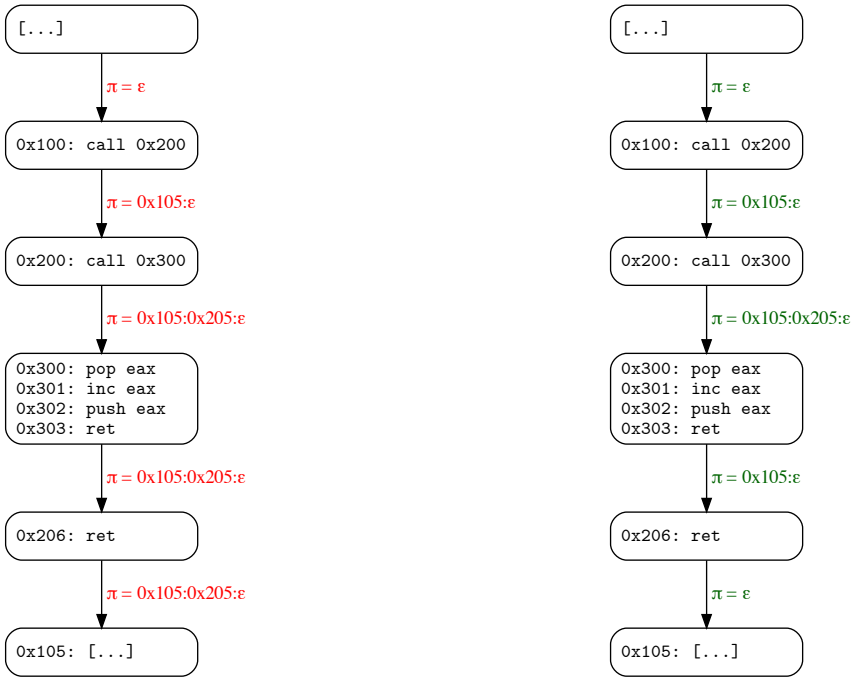


FIGURE 6.2 – Exemple d'erreur sur la détection des falsifications de la pile d'appels (1)

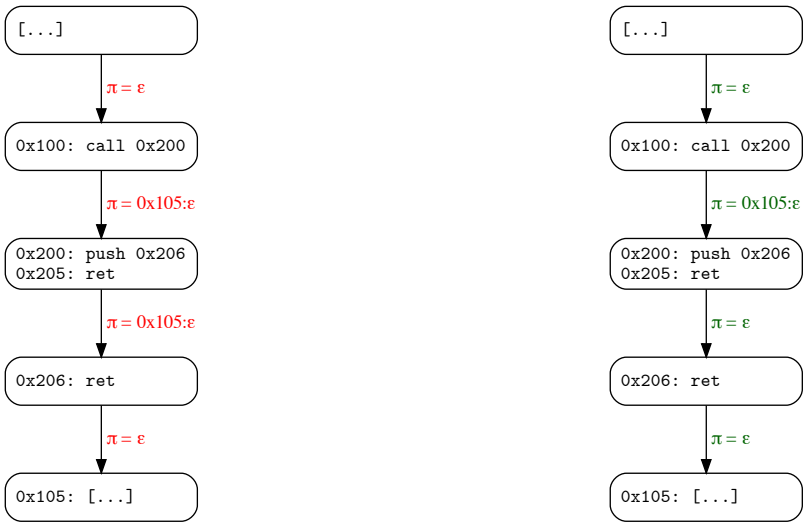


FIGURE 6.3 – Exemple d'erreur sur la détection des falsifications de la pile d'appels (2)

6.4.2 Adresse de retour des `CALL`

Dans BOA, nous faisons l'hypothèse que les octets présents en aval des instructions `CALL` ne correspondent pas à du code valide, de ce fait, par défaut, nous ne les désassemblons pas. Afin de désassembler l'adresse *return-site* d'un `CALL` nous devons au minimum trouver un chemin d'exécution menant à cette adresse.

Cette précaution nous permet d'éviter de désassembler du code mort ou du code invalide dans les cas où la fonction appelée ne revient pas, notamment dans le cas d'une falsification de la pile d'appels.

6.4.3 Conclusion sur le statut d'un `RET`

Finalement, le statut d'un `RET` est donné lorsque BOA a terminé la construction du graphe de flot de contrôle. Un `RET` sera dit authentique s'il revient bien au *return-site* attendu pour tous les tours de boucle de BOA. De la même façon, un `RET` sera dit falsifié s'il ne revient pas au bon *return-site* pour tous les tours de boucle de BOA. Dans tous les autres cas, il sera dit inconnu.

6.5 Expériences

Nous présentons dans cette section les deux expériences que nous avons effectuées afin de mettre en évidence les résultats que BOA obtient concernant le calcul des adresses cibles des sauts indirects ainsi que la détection des falsifications de la pile d'appels.

6.5.1 Validation de BOA

BOA est principalement destiné au désassemblage de logiciels malveillants embarquant pour la plupart différentes techniques d'anti-analyse comme des obfuscations. Cependant, nous souhaitons vérifier dans un premier temps que BOA est capable de désassembler correctement des binaires non obfusqués et compilés avec un compilateur standard. Pour ces binaires non protégés, nous souhaitons notamment vérifier que BOA :

1. Est capable d'effectuer un désassemblage correct et complet ;
2. Parvient à résoudre les adresses cibles des instructions de saut indirect (`CALL`, `JMP`, `RET`) ;
3. Ne détecte aucune falsification de la pile d'appels.

6.5.1.1 Méthode

Jeu de programmes utilisé Pour répondre à cette question, nous utilisons les programmes du groupe Mälardalen WCET. Ce groupe de recherche propose un jeu de programmes en langage C initialement utilisé pour évaluer des outils d'analyse WCET (*Worst-Case Execution Time*) [3]. Pour plusieurs de nos expériences, nous avons choisi d'utiliser ces programmes car ils ont les avantages suivants :

— Nous disposons du code source.
— Ils contiennent des structures variées (*for*, *if*, *while*, *switch*, plusieurs fonctions).
— Ils sont simples à compiler (pas de fichier *header*).
— Ils ne nécessitent aucune bibliothèque externe.
— Ils contiennent leurs propres données d'entrées (inutile de donner des arguments lors de l'exécution) ce qui signifie que lors de l'exécution de ces programmes un seul chemin d'exécution est possible.

Le dernier point nous est particulièrement utile car il nous permet de faire le postulat suivant : comme un seul chemin d'exécution est possible lors de l'exécution de chaque programme, cela signifie qu'il nous suffit d'exécuter chaque programme une seule fois afin d'obtenir l'unique trace d'exécution contenant l'ensemble du code « vivant » du programme. De plus, cette trace d'exécution nous donnera, pour chacune des instructions de saut indirect du programme, l'ensemble de ses adresses cibles.

3. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

Il est nécessaire d'indiquer que nous avons retiré de notre jeu de test tous les programmes effectuant des opérations sur les *float* car la platefrome BINSEC que nous utilisons pour la traduction en langage DBA des instructions x86 ne gère pas encore ce type d'instructions.

Construction du jeu de binaires Pour obtenir les binaires exécutables à soumettre à BOA pour cette expérience, nous avons compilé chaque programme C en exécutables Linux 32 bits avec les options de compilation `-O0` à `-O3`. Une fois les binaires contenant des instructions sur les nombres flottants retirés, nous obtenons un jeu de 96 binaires.

Nous avons ensuite tracé chaque binaire de ce jeu à l'aide d'un *PinTool* afin de générer une trace d'exécution nous permettant de récupérer, comme indiqué précédemment, la liste des instructions assembleurs uniques de chaque programme ainsi que la liste des adresses destinations de chaque instruction de saut indirect. Nous aurions également pu utiliser IDA afin de récupérer la liste des instructions de chaque binaire, cependant IDA ne permet pas d'obtenir les adresses de saut des instructions de saut dynamique. Finalement nous avons désassemblé chaque binaire avec BOA.

Pour répondre à la première question, nous vérifions que les instructions désassemblées par BOA sont les mêmes que celles présentes dans la trace d'exécution. Si BOA en désassemble plus ou qu'il désassemble des instructions non présentes dans la trace d'exécution, cela signifie que le désassemblage de BOA n'est pas correct et si BOA ne désassemble pas toutes les instructions vues dans la trace d'exécution cela signifie que le désassemblage de BOA n'est pas complet.

Concernant les instructions de saut dynamique, nous utilisons la trace afin de récupérer la liste des adresses de saut de chaque instruction. Nous pouvons alors vérifier si BOA a réussi à calculer les adresses de sauts vues dans la trace.

Enfin, pour les falsifications de la pile d'appels, il nous suffit encore une fois de vérifier que dans la trace cette obfuscation n'apparaît pas (ce qui est normalement le cas avec des binaires compilés avec GCC) et de vérifier que BOA n'en détecte pas non plus lors de son analyse.

6.5.1.2 Résultats

Le premier tableau de résultats (6.1) permet de montrer la correction et la complétude du désassemblage de BOA. Pour chaque binaire, nous comparons les instructions que nous avons trouvées dans la trace avec celles que BOA a pu désassembler. Dans notre cas, BOA a été capable, pour l'ensemble des binaires de cette expérience, de retrouver les mêmes instructions que celles présentes dans la trace. Notons également que le temps moyen d'analyse pour cette expérience est de 14 secondes par binaire. Nous en concluons alors que le désassemblage effectué par BOA est complet et correct sur des binaires non obfusqués.

Le deuxième tableau de résultats (6.2) concerne exclusivement l'étude des instructions `RET`. BOA a été capable de calculer la totalité des adresses de saut des `RET` vues dans la trace. De plus, BOA n'a détecté aucune falsification de la pile d'appels.

Le dernier tableau de résultats (6.3) présente les résultats concernant les instructions de saut dynamique autre que les `RETs`. Cependant les programmes WCET utilisent peu ces instructions. Les seuls cas où une instruction de saut dynamique est utilisée concernent les programmes utilisant un *switch*. Nous proposons alors dans ce tableau seulement les binaires dans lesquels au moins une instruction de saut dynamique est utilisée. Malgré le faible échantillon de binaires pour cette expérience, nous pouvons voir que BOA a été capable de calculer l'ensemble des adresses de saut vues dans la trace d'exécution.

6.5.1.3 Conclusion

Ces résultats nous montrent que le désassemblage de BOA est correct et complet lorsque le binaire désassemblé n'est pas obfusqué. De plus, BOA est parfaitement capable de calculer les adresses de saut des instructions `RET` et des instructions de saut dynamique. Finalement dans ce contexte et pour les questions posées, BOA est capable d'obtenir les mêmes résultats qu'une analyse dynamique à base d'instrumentation (*PinTool* ou *Dynamorio*) mais en utilisant une approche entièrement statique et avec pour seule information d'entrée le binaire à désassembler.

TABLE 6.1 – Désassemblage des programmes WCET originaux

Binaire	**Trace**	**BOA**	**Temps** (mm:ss,ms)	**Binaires**	**Trace**	**BOA**	**Temps** (mm:ss,ms)
adpcm O0	1613	1613 ✓	01:02,317	fir O0	110	110 ✓	03:16,640
adpcm O1	1030	1030 ✓	00:20,330	fir O1	75	75 ✓	01:19,629
adpcm O2	1127	1127 ✓	00:21,300	fir O2	77	77 ✓	01:16,379
adpcm O3	1775	1775 ✓	00:20,117	fir O3	77	77 ✓	01:15,590
bs O0	62	62 ✓	00:00,806	insertsort O0	69	69 ✓	00:00,940
bs O1	38	38 ✓	00:00,739	insertsort O1	38	38 ✓	00:00,588
bs O2	25	25 ✓	00:00,594	insertsort O2	44	44 ✓	00:00,619
bs O3	25	25 ✓	00:00,647	insertsort O3	73	73 ✓	00:00,954
cnt O0	189	189 ✓	00:03,679	janne_complex O0	47	47 ✓	00:00,958
cnt O1	107	107 ✓	00:02,220	janne_complex O1	32	32 ✓	00:00,760
cnt O2	102	102 ✓	00:01,466	janne_complex O2	32	32 ✓	00:01,500
cnt O3	306	306 ✓	00:01,815	janne_complex O3	33	33 ✓	00:00,792
compress O0	315	315 ✓	00:03,704	jfdctint O0	485	485 ✓	00:05,169
compress O1	235	235 ✓	00:03,434	jfdctint O1	215	215 ✓	00:01,920
compress O2	243	243 ✓	00:03,568	jfdctint O2	219	219 ✓	00:01,855
compress O3	236	236 ✓	00:03,375	jfdctint O3	219	219 ✓	00:02,616
cover O0	446	446 ✓	00:10,746	lcdnum O0	52	52 ✓	00:00,871
cover O1	428	428 ✓	00:10,433	lcdnum O1	40	40 ✓	00:00,763
cover O2	323	323 ✓	00:07,693	lcdnum O2	29	29 ✓	00:00,455
cover O3	307	307 ✓	00:07,163	lcdnum O3	55	55 ✓	00:00,760
crc O0	166	166 ✓	00:10,174	matmult O0	203	203 ✓	02:16,823
crc O1	110	110 ✓	00:06,306	matmult O1	108	108 ✓	00:42,857
crc O2	114	114 ✓	00:06,720	matmult O2	139	139 ✓	00:40,342
crc O3	141	141 ✓	00:04,140	matmult O3	112	112 ✓	00:40,664
duff O0	145	145 ✓	00:01,833	ndes O0	696	696 ✓	00:42,165
duff O1	98	98 ✓	00:01,118	ndes O1	481	481 ✓	00:29,629
duff O2	89	89 ✓	00:00,946	ndes O2	555	555 ✓	00:15,769
duff O3	89	89 ✓	00:01,210	ndes O3	724	724 ✓	00:19,148
edn O0	843	843 ✓	00:42,278	ns O0	67	67 ✓	00:07,118
edn O1	468	468 ✓	00:19,129	ns O1	54	54 ✓	00:03,436
edn O2	436	436 ✓	00:20,105	ns O2	42	42 ✓	00:06,872
edn O3	421	421 ✓	00:20,170	ns O3	72	72 ✓	00:04,819
expint O0	104	104 ✓	00:02,346	nsichneu O0	3511	3511 ✓	00:38,728
expint O1	74	74 ✓	00:01,901	nsichneu O1	2770	2770 ✓	00:37,506
expint O2	34	34 ✓	00:01,882	nsichneu O2	2398	2398 ✓	00:33,473
expint O3	34	34 ✓	00:00,759	nsichneu O3	2398	2398 ✓	00:28,320
fac O0	47	47 ✓	00:00,943	prime O0	97	97 ✓	00:04,403
fac O1	48	48 ✓	00:01,950	prime O1	43	43 ✓	00:01,576
fac O2	30	30 ✓	00:00,491	prime O2	37	37 ✓	00:01,568
fac O3	30	30 ✓	00:00,551	prime O3	32	32 ✓	00:01,577
fdct O0	590	590 ✓	00:05,350	recursion O0	58	58 ✓	00:04,169
fdct O1	228	228 ✓	00:02,359	recursion O1	44	44 ✓	00:02,730
fdct O2	228	228 ✓	00:01,590	recursion O2	46	46 ✓	00:01,824
fdct O3	228	228 ✓	00:01,775	recursion O3	182	182 ✓	00:02,357
fibcall O0	39	39 ✓	00:01,178	statemate O0	684	684 ✓	00:03,610
fibcall O1	25	25 ✓	00:00,646	statemate O1	556	556 ✓	00:03,199
fibcall O2	12	12 ✓	00:00,346	statemate O2	700	700 ✓	00:03,837
fibcall O3	12	12 ✓	00:00,428	statemate O3	630	630 ✓	00:03,220

Trace : nombre d'instructions uniques dans la trace d'exécution
BOA : nombre d'instructions uniques désassemblées par BOA

TABLE 6.2 – Analyse des instructions `RET` des programmes WCET originaux

Binaire	**# `RET`**	**# Cibles max**	**Complet / Non falsifié**	**Binaires**	**# `RET`**	**# Cibles max**	**Complet / Non falsifié**
adpcm O0	19	11	✓/✓	fir O0	2	2	✓/✓
adpcm O1	17	4	✓/✓	fir O1	2	1	✓/✓
adpcm O2	9	4	✓/✓	fir O2	2	1	✓/✓
adpcm O3	6	2	✓/✓	fir O3	2	1	✓/✓
bs O0	2	2	✓/✓	insertsort O0	1	1	✓/✓
bs O1	2	1	✓/✓	insertsort O1	1	1	✓/✓
bs O2	1	1	✓/✓	insertsort O2	1	1	✓/✓
bs O3	1	1	✓/✓	insertsort O3	1	1	✓/✓
cnt O0	9	2	✓/✓	janne_complex	2	2	✓/✓
cnt O1	5	3	✓/✓	janne_complex	1	1	✓/✓
cnt O2	3	3	✓/✓	janne_complex	0	0	✓/✓
cnt O3	2	2	✓/✓	janne_complex	0	0	✓/✓
compress O0	6	3	✓/✓	jfdctint O0	3	1	✓/✓
compress O1	8	2	✓/✓	jfdctint O1	2	2	✓/✓
compress O2	4	2	✓/✓	jfdctint O2	2	2	✓/✓
compress O3	3	1	✓/✓	jfdctint O3	2	2	✓/✓
cover O0	4	4	✓/✓	lcdnum O0	3	1	✓/✓
cover O1	5	2	✓/✓	lcdnum O1	3	1	✓/✓
cover O2	5	1	✓/✓	lcdnum O2	1	1	✓/✓
cover O3	5	1	✓/✓	lcdnum O3	1	1	✓/✓
crc O0	4	2	✓/✓	matmult O0	8	4	✓/✓
crc O1	3	2	✓/✓	matmult O1	5	2	✓/✓
crc O2	3	2	✓/✓	matmult O2	2	2	✓/✓
crc O3	3	2	✓/✓	matmult O3	2	2	✓/✓
duff O0	5	1	✓/✓	ndes O0	6	9	✓/✓
duff O1	3	2	✓/✓	ndes O1	7	9	✓/✓
duff O2	2	2	✓/✓	ndes O2	5	3	✓/✓
duff O3	2	2	✓/✓	ndes O3	5	3	✓/✓
edn O0	10	8	✓/✓	ns O0	3	1	✓/✓
edn O1	8	1	✓/✓	ns O1	2	1	✓/✓
edn O2	6	1	✓/✓	ns O2	1	1	✓/✓
edn O3	6	1	✓/✓	ns O3	1	1	✓/✓
expint O0	3	3	✓/✓	nsichneu O0	1	1	✓/✓
expint O1	2	1	✓/✓	nsichneu O1	1	1	✓/✓
expint O2	0	0	✓/✓	nsichneu O2	1	1	✓/✓
expint O3	0	0	✓/✓	nsichneu O3	1	1	✓/✓
fac O0	2	2	✓/✓	prime O0	5	5	✓/✓
fac O1	1	2	✓/✓	prime O1	1	2	✓/✓
fac O2	0	0	✓/✓	prime O2	1	2	✓/✓
fac O3	0	0	✓/✓	prime O3	0	0	✓/✓
fdct O0	3	1	✓/✓	recursion O0	3	3	✓/✓
fdct O1	2	1	✓/✓	recursion O1	2	3	✓/✓
fdct O2	2	1	✓/✓	recursion O2	2	2	✓/✓
fdct O3	2	1	✓/✓	recursion O3	2	2	✓/✓
fibcall O0	2	2	✓/✓	statemate O0	9	7	✓/✓
fibcall O1	1	1	✓/✓	statemate O1	9	6	✓/✓
fibcall O2	0	0	✓/✓	statemate O2	8	5	✓/✓
fibcall O3	0	0	✓/✓	statemate O3	6	3	✓/✓

`RET` : nombre d'instructions `RET` désassemblées par BOA
Cibles max : nombre d'adresses de saut différentes pour le `RET` qui en pocède le plus
Complet : si toutes les adresses de saut de tous les `RET` ont été trouvées par BOA
Non falsifié : si tous les `RET` ont été détectés comme non falsifiés par BOA

Table 6.3 – Analyse des instructions de saut indirect des programmes WCET originaux

Binaire	# Sauts dynamiques	# Cibles max	Complet
cover O0	3	120	✓
cover O1	3	120	✓
cover O2	2	120	✓
cover O3	2	120	✓
duff O0	1	1	✓
duff O1	1	1	✓
duff O2	1	1	✓
duff O3	1	1	✓
lcdnum O0	1	1	✓
lcdnum O1	1	1	✓

Sauts dynamiques : nombre d'instructions de saut dynamique (hors `RET`) désassemblées par BOA
Cibles max : nombre d'adresses de saut différentes pour le saut dynamique en ayant le plus
Complet : si toutes les adresses de saut de tous les sauts dynamiques ont été trouvées par BOA

6.5.2 Binaires protégés

Dans un second temps, nous souhaitons mesurer l'efficacité de BOA face à des binaires obfusqués. Pour cette expérience nous cherchons à savoir si BOA est capable de correctement calculer les adresses de saut des instructions de saut dynamique dans des codes protégés. Plus précisément, nous souhaitons savoir si les résultats de BOA pour cette fonctionnalité sont incomplets (si BOA ne parvient pas à calculer toutes les adresses de saut d'une instruction de saut) ou bien au contraire si les résultats de BOA comportent des erreurs (si BOA calcule des adresses de saut fausses). De plus, nous voulons montrer que BOA est capable de détecter les obfuscations de type falsification de la pile d'appels.

6.5.2.1 Jeux de binaires utilisés

Pour cette expérience nous utilisons deux jeux de binaires différents que nous décrivons dans cette section.

Programmes Mälardalen WCET obfusqués par Tigress Pour construire notre premier jeu de binaires, nous utilisons les programmes du groupe Mälardalen WCET que nous protégeons à l'aide de l'obfuscateur de code source Tigress [4].

Cet outil permet d'obfusquer le code source d'un programme C en y ajoutant différentes « transformations ». Les différentes options sont directement données en arguments et permettent par exemple de demander à Tigress de découper les fonctions du programme à transformer ou au contraire de fusionner plusieurs fonctions.

Pour cette expérience nous appliquons la transformation *Encode Branches* à chaque programme du jeu Mälardalen WCET. Cette transformation permet de remplacer la plupart des sauts statiques par des sauts dynamiques en utilisant notamment l'instruction `RET` de façon détournée. Ce type de transformation dégrade généralement le désassemblage effectué de façon statique.

Packers commerciaux Windows Un binaire packé est un excellent candidat à BOA car il permet de soumettre BOA à une multitude d'obfuscations présentes au sein d'un seul et même binaire.

Pour construire notre jeu de binaires, nous avons packé le binaire *hostname.exe* de Windows à l'aide des différents packers Windows que nous avons en notre possession.

4. https://tigress.wtf

L'avantage d'utiliser *hostname* pour cette expérience est de pouvoir rapidement vérifier si l'opération de package a fonctionné. Pour cela il suffit d'exécuter chaque binaire généré et de vérifier que le nom de la machine apparaît bien dans le terminal.

En utilisant un *PinTool* résistant aux obfuscations les plus courantes et basé sur un modèle de vagues afin de gérer les auto-modifications, nous avons pu générer la trace d'exécution de chacun des binaires générés. Ces traces d'exécution nous servent de support afin de vérifier les résultats obtenus grâce à BOA, notamment afin de repérer les différentes obfuscations mais aussi les adresses de changement de vague. Cependant il est important de rappeler que lors de son analyse, BOA n'utilise pas ces traces d'exécution.

6.5.2.2 Méthode

Pour cette expérience nous utilisons les deux jeux de binaires que nous venons de décrire ; les binaires du groupe Mälardalen WCET obfusqués à l'aide de la transformation *Encode Branches* de Tigress ainsi que les binaires *hostname* packés. Pour les binaires *hostname*, nous avons fait le choix d'effectuer cette expérience uniquement sur la première vague d'auto-modification car ce n'est pas cette obfuscation que nous souhaitons mettre en évidence ici.

L'avantage de ces deux jeux de binaires est le fait qu'un seul chemin d'exécution soit possible. Cela nous permet de comparer les résultats que BOA obtient avec les résultats que l'on obtient dans la trace d'exécution.

Durant l'expérience nous vérifions plusieurs points. Nous regardons d'abord que BOA n'a pas fait de sur ou sous-désassemblage : pour cela nous comparons les instructions découvertes par BOA avec celles présentes dans la trace d'exécution. Ensuite nous vérifions pour chaque instruction de saut indirect que BOA a réussi à calculer l'ensemble des adresses de saut (pas plus et pas moins). Pour cela nous prenons la trace d'exécution pour référence. Enfin, concernant les falsifications de la pile d'appels, nous comparons encore une fois les résultats donnés par BOA avec ceux que l'on peut observer dans la trace d'exécution.

6.5.2.3 Résultats

Le tableau 6.4 donne les résultats des binaires du groupe Mälardalen WCET obfusqués. Cette expérience montre un résultat de 100% pour BOA ; nous obtenons un désassemblage complet et correct, les adresses de saut calculées sont identiques à celles rencontrées dans la trace et enfin la détection des falsifications de la pile d'appels est en accord avec la détection effectuée sur la trace.

Le tableau 6.5 donne les résultats que l'on obtient pour la détection de falsification de la piles d'appels de la première vague des différents packers de notre jeu. On remarque que BOA a été capable pour chaque binaire d'obtenir un résultat identique au résultat obtenu grâce à la trace d'exécution.

6.5.2.4 Conclusion

Cette expérience nous montre que BOA obtient des résultats tout à fait intéressants concernant le calcul des adresses de saut des instructions de saut indirect. La détection de falsification de la pile d'appels fonctionne également correctement. Notons tout de même que cette détection dépend notamment du résultat du calcul de l'adresse de saut du `RET`. Si pour un `RET`, donné BOA n'a pas été en mesure de calculer son adresse de saut alors il lui est impossible d'effectuer la détection de falsification de la pile d'appels.

Rappelons également que les binaires utilisés pour cette expérience ne nécessitent pas d'argument d'entrée. Comme nous l'avons expliqué dans la section 6, les adresses de saut des instructions de saut dépendantes des entrées utilisateurs ne peuvent être calculées par BOA. Nous notons alors qu'en condition réelle il est bien plus probable que BOA produise un résultat incomplet plutôt qu'incorrect.

TABLE 6.4 – Exéprience WCET *Encode Branches*

Binaire	**Trace**	**BOA**	**`RET`** NF/F/cibles max	**Sauts dyn.** total/cibles max	**Temps** (mm:ss,ms)
bs	346	346 ✓	5/9/4	3/2	00:03,995
cnt	988	988 ✓	12/13/7	7/5	00:17,535
compress	1331	1331 ✓	54/33/4	6/7	00:20,749
crc	687	687 ✓	12/20/6	4/3	00:41,111
duff	677	677 ✓	7/8/2	5/8	00:05,738
edn	2571	2571 ✓	14/33/13	10/4	01:08,390
expint	519	519 ✓	7/13/5	4/4	00:05,670
fac	321	321 ✓	3/6/1	3/4	00:03,202
fdct	983	983 ✓	6/5/1	2/5	00:05,549
fibcall	346	346 ✓	3/3/1	2/5	00:02,346
insertsort	239	239 ✓	4/5/1	2/3	00:02,587
janne_complex	398	398 ✓	7/12/2	3/6	00:03,387
jfdctint	840	840 ✓	8/9/2	3/5	00:06,391
lcdnum	356	356 ✓	6/6/1	3/6	00:03,118
ndes	2877	2877 ✓	45/59/16	6/14	01:36,830
ns	1647	1647 ✓	10/13/3	3/6	00:13,552
prime	754	754 ✓	7/10/2	6/6	00:23,241
Moyenne					00:19,023

Trace : nombre d'instructions uniques dans la trace d'exécution
BOA : nombre d'instructions uniques désassemblées par BOA
(NF/F/cibles max) : (nombre de `RET` non falsifiés/nombre de `RET` falsifiés/nombre d'adresses de saut pour le `RET` qui en pocède le plus)
(total/cibles max) : (nombre d'instructions uniques de type saut dynamique (or `RET`)/nombre d'adresses de saut différentes pour le saut dynamique qui en pocède le plus)

TABLE 6.5 – Exéprience `RET` packers

Packer	**# Non falsifiés** BOA/trace	**# Falsifiés** BOA/trace	**# Cibles max**
ACPacker	0/0	19/19	1
ACProtect 2.0	0/0	2/2	1
ASPack 2.40	6/6	5/5	5
ASProtect SKE 2.51	1/1	2/2	1
Enigma Protector 4.30	0/0	0/0	
EP Protector 0.3	0/0	0/0	
FSG 2.0	2/2	0/0	6
Morphine 2.7	0/0	0/0	
JDPack 2.0	0/0	0/0	
Mew	2/2	1/1	6
MoleBox	5/5	0/0	4
Mystic	7/7	0/0	76
nPack 1.1.300	17/17	1/1	6
Obsidium 1.3.6.4	0/0	1/1	1
Packman 1.0	3/3	0/0	6
PE Compact 2.20	0/0	0/0	
PE Compact 3.02.2	0/0	0/0	
PELock	0/0	0/0	
PESpin 1.1	0/0	0/0	
Petite 2.2	4/4	0/0	4
RLPack	10/10	0/0	6
SVK 1.43f	0/0	0/0	
tElock 0.51	0/0	0/0	
tElock 0.99	0/0	0/0	
Themida 1.8	1/1	0/0	1
Upack 0.39	2/2	0/0	2
UPX 2.90	0/0	0/0	
WinUpack	3/3	0/0	11
Yoda's Crypter 1.3	0/0	0/0	
Yoda's Protector 1.02	1/1	1/1	1

Cibles max : nombre d'adresses de saut différentes pour le `RET` qui en pocède le plus

Chapitre 7

Prédicats opaques et branches mortes

7.1 Objectifs

Nous avons vu dans la section 2.2.3 qu'une des principales conséquences des prédicats opaques est le désassemblage par erreur de code mort en aval des branches mortes.

Nous souhaitons dans BOA éviter au maximum de désassembler du code mort pour obtenir un graphe de flot de contrôle le plus correct possible. Notre objectif n'est donc pas de détecter uniquement des prédicats opaques mais de façon plus générale de repérer les branches mortes.

7.2 État de l'art

Collberg et al. [21, 22] proposent les premières définitions et classifications des prédicats opaques. Cette obfuscation utilise différentes structures et propriétés afin d'arriver à ses fins. On peut par exemple citer les invariances de données, les invariances arithmétiques ou encore les invariances dépendantes du système (en utilisant le résultat d'un appel à fonction de l'API Windows par exemple). De plus, les prédicats sont souvent difficiles à résoudre car ils peuvent être basés sur des formules de type *Mixed Boolean Arithmetic* (MBA) [77] ou encore des fonctions cryptographiques [66].

Banescu et al. [6] montrent que l'exécution symbolique (ici avec l'utilisation de KLEE [18]) s'en sort particulièrement bien lorsqu'il s'agit de détecter des prédicats opaques générés à l'aide de l'obfuscateur de code source Tigress.

Bardin, David et Marion [8] montrent également qu'il est possible d'effectuer la détection des prédicats opaques en utilisant une technique d'exécution symbolique en arrière de longueur bornée basée sur une trace d'exécution.

Quant à eux, Preda et al. [55] ont implémenté une méthode de détection des prédicats opaques basée sur une de l'interprétation abstraite.

7.3 Traitement dans BOA

Dans BOA nous avons fait le choix de mettre en place deux approches différentes afin de détecter et gérer les prédicats opaques ainsi que les branches mortes.

La première approche concerne les branches mortes ainsi que le code mort de façon plus générale. Cette gestion est intrinsèquement liée à la façon dont fonctionne notre algorithme de construction du graphe de flot de contrôle d'un binaire.

La seconde approche est une analyse à part entière qui est plus orientée sur la détection des prédicats opaques. Cette analyse nécessite que le graphe de flot de contrôle du binaire à analyser soit déjà construit.

7.3.1 Gestion des branches mortes et code mort

Durant l'exécution de sa boucle principale, lorsque BOA traite un bloc de base ayant pour dernière instruction un saut conditionnel, il va tenter de calculer la valeur concrète de la condition en fonction de l'état machine actuel. Si la valeur de la condition est concrète pour ce tour de boucle alors BOA va seulement explorer la branche correspondant à la valeur de la condition lors du prochain tour de boucle.

Finalement, une fois la construction du graphe de flot terminée, si un `Jcc` n'a qu'un seul arc successeur dans le graphe alors cela signifie que pour tous les chemins d'exécution que BOA a empruntés, la valeur de la condition évaluée par ce saut conditionnel était toujours concrète et de même valeur. Par conséquent, la seconde branche qui n'apparaît pas dans le graphe est considérée comme une branche morte. D'un autre coté, si aucune autre instruction de saut ne permet d'atteindre l'adresse de la branche morte, alors BOA a évité de désassembler du code mort.

À ce stade, concernant les `Jcc` pour lesquels une seule branche est présente dans le graphe de flot de contrôle, BOA n'est pas capable de se prononcer quant au fait que ces sauts conditionnels soient des prédicats opaques ou non. La seule chose que l'on puisse dire c'est que BOA n'a trouvé aucun chemin d'exécution valide menant à leurs branches mortes.

7.3.2 Détection des prédicats opaques

Parallèlement à la gestion des branches mortes effectuée par BOA durant la construction du graphe de flot de contrôle, BOA propose une analyse permettant la détection des prédicats opaques dans un graphe de flot de contrôle existant.

Concrètement, une option permet d'effectuer cette analyse directement sur le graphe de flot de contrôle généré par BOA, cependant il est possible d'adapter BOA afin qu'il puisse effectuer cette analyse sur un graphe de flot de contrôle provenant d'un autre outil.

L'analyse va chercher à calculer, pour chaque instruction de saut conditionnel du graphe, si la valeur de la condition évaluée par celui-ci est concrète et unique, et ce, pour n'importe quel état machine.

Le fonctionnement de notre analyse est basé sur l'hypothèse affirmant que l'ensemble des éléments nécessaires à la création d'un prédicat opaque (données et instructions) appartiennent tous au bloc de base dans lequel se trouve l'instruction de saut conditionnel concernée. Du fait de cette hypothèse, notre analyse ne sera pas en mesure de détecter les prédicats opaques construits sur plusieurs blocs de base.

Finalement, pour savoir si le saut conditionnel d'un bloc de base est un prédicat opaque, nous cherchons à savoir si la valeur de la condition évaluée par cette instruction est concrète et unique pour n'importe quel état machine en entrée du bloc de base. Afin de répondre à cette question nous effectuons l'exécution symbolique du bloc de base avec un état machine d'entrée pour lequel tous les éléments de la machine (registres et cases mémoire) sont inconnus (dont la valeur est $\perp$). À l'issue de cette exécution symbolique, si la valeur du pointeur d'instruction $\texttt{ip}'$ est concrète alors cela signifie que le bloc de base contient à lui seul l'ensemble des informations nécessaires à fixer la valeur de la condition évaluée par le saut conditionnel et donc que ce dernier est un prédicat opaque.

Enfin, il suffit d'appliquer cette méthode sur l'ensemble des blocs de base ayant pour dernière instruction un saut conditionnel afin de connaître les prédicats opaques du graphe de flot de contrôle.

7.4 Expériences

Comme nous l'avons précédemment expliqué, la présence d'un prédicat opaque conduit nécessairement à l'apparition d'une branche morte alors que la présence d'une branche morte dans un binaire n'est pas nécessairement causée par un prédicat opaque.

Dans notre cas, nous avons pour objectif principal de ne pas désassembler les branches mortes d'un saut conditionnel, qu'il soit causé par un prédicat opaque ou non. C'est pour cela que l'algorithme de BOA permettant la construction du graphe de flot de contrôle d'un binaire est implémenté de façon à respecter si possible cet objectif.

BOA embarque une analyse permettant la détection des prédicats opaques. Cette analyse doit être demandée explicitement par l'utilisateur et est complètement indépendante de la phase de désassemblage et de construction du graphe de flot de contrôle car effectuée dans un second temps.

Nous souhaitons montrer ici que BOA est à la fois capable d'éviter une exploration et un désassemblage à tort du code mort présent en aval d'une branche morte mais aussi que son analyse permettant la détection des prédicats opaques est efficace.

7.4.1 Méthode

Nous effectuons ici deux expériences avec deux jeux de binaires différents.

La première expérience nous a permis de vérifier si BOA est capable de ne pas désassembler les branches mortes d'un programme. Nous utilisons pour cela les binaires du groupe Mälardalen WCET obfusqués à l'aide de la transformation *Add Opaque*. Nous souhaitons montrer que le désassemblage de BOA permet de retrouver les mêmes instructions que celles que l'on peut retrouver dans la trace d'exécution de chaque binaire. Si BOA ne désassemble aucune instruction supplémentaire que celles que l'on trouve dans la trace alors cela signifie que BOA n'a emprunté et désassemblé aucune branche morte.

Pour la seconde expérience, nous souhaitons montrer que l'analyse de BOA permettant la détection des prédicats opaques est efficace. Pour cela nous appliquons cette analyse sur la logiciel malveillant XTunnel [5] précédemment désassemblé (désassemblage classique statique récursif) par BOA. Ce binaire est connu pour contenir beaucoup de prédicats opaques et nous pourrons comparer nos résultats avec ceux obtenus par David [25] et Salwan [63] ayant utilisé respectivement les outils BINSEC et Triton.

7.4.2 Résultats

Les résultats de la première expérience sur les binaires WCET sont donnés dans le tableau 7.1. Nous pouvons constater que pour les binaires utilisés lors de cette expérience, le résultat pour BOA est de 100% car malgré la présence des branches mortes ajoutées par Tigress, BOA a été capable de retrouver les mêmes instructions que celles obtenues dans la trace d'exécution, pas plus et pas moins.

Concernant la détection des prédicats opaques dans le binaire XTunnel, les résultats sont donnés dans le tableau 7.2. Sur le nombre de conditions analysées nous obtenons un total similaire à celui de BINSEC alors que Triton semble analyser environs 20 000 conditions de plus que nous. À propos des résultats, nous remarquons que le nombre de prédicats opaques détectés par BOA est très similaire à celui obtenu par David [25] et Salwan [63]. Notons cependant que le nombre de cas marqués comme inconnus pour BOA est largement inférieur à celui de BINSEC. Enfin, le temps de calcul nécessaire à BOA est similaire à celui nécessaire par Triton alors que BINSEC demande en moyenne cinq fois plus de temps.

7.4.3 Conclusion

Concernant les branches mortes, les résultats nous montrent que BOA est capable de les éviter correctement, réduisant ainsi le risque de désassemblage de code mort ou de *junk code*.

Pour la détection des prédicats opaques, nous avons vu dans cette section que BOA effectue son calcul au seul niveau du bloc de base contenant le saut conditionnel à analyser. Or, nous remarquons que les résultats de BOA sont assez similaires à ceux obtenus par les outils BINSEC et Triton. Cela nous montre que les instructions intervenant dans l'expression d'un prédicat opaque sont généralement présentes dans le bloc de base accueillant le saut conditionnel de ce prédicat.

5. SHA1 : `99b454262dc26b081600e844371982a49d334e5e`

TABLE 7.1 – Expérience WCET *Add Opaque*

Binaires	**Trace**	**BOA**	**Temps** (mm:ss,ms)
bs	352	352 ✓	00:03,530
cnt	812	812 ✓	00:12,691
compress	1124	1124 ✓	00:26,294
cover	819	819 ✓	00:13,312
crc	609	609 ✓	00:32,875
duff	492	492 ✓	00:04,896
edn	2169	2169 ✓	01:21,408
expint	400	400 ✓	00:04,200
fac	270	270 ✓	00:03,144
fdct	960	960 ✓	00:06,597
fibcall	265	265 ✓	00:03,299
fir	3237	3237 ✓	04:37,900
insertsort	183	183 ✓	00:02,156
janne_complex	276	276 ✓	00:02,326
jfdctint	720	720 ✓	00:06,329
lcdnum	299	300 ✓	00:03,252
matmult	811	811 ✓	02:52,995
ndes	2272	2272 ✓	01:41,883
ns	2760	2760 ✓	00:13,810
nsichneu	3939	3939 ✓	00:32,386
prime	603	603 ✓	00:23,499
recursion	284	284 ✓	00:14,703
Moyenne			00:38,340

Trace : nombre d'instructions uniques dans la trace d'exécution
BOA : nombre d'instructions uniques désassemblées par BOA

TABLE 7.2 – Exéprience XTunnel

Plateforme	**Conditions analysées**	**Non opaques**	**Opaques**	**Inconnues**	**Temps**	**Temps moyen pour 100 conditions**
Triton	50302	43093	7209	?	23 min	2,74 sec
BinSec	30147	17062 *(914 FN)*	12333 *(2543 FP)*	652	50 min 59	10,15 sec
BOA	29973	22255	7562 *(5333 TF)* *(2229 TV)*	156	15 min 23(†) 9 min 15(††)	3,08 sec(†) 1,85 sec(††)

FN : faux négatifs, FP : faux positifs
TV : toujours vraie, TF : toujours fausse
(†) : MacBook Pro i7, 4 threads
(††) : Grid'5000, 32 threads

Chapitre 8

Auto-modification et gestion des vagues

8.1 Notion de vagues

Comme nous l'avons expliqué dans la section 2.2.4, l'auto-modification consiste à modifier la valeur d'une ou plusieurs cases mémoire avant de transférer l'exécution sur ces adresses, menant ainsi le processeur à exécuter des instructions dont l'opcode n'était pas présent lors du chargement du binaire.

Rien n'empêche ces « nouvelles » instructions, durant leur exécution, de ne pas elles-même générer une auto-modification. On remarque alors qu'une notion de vague apparaît et qu'un code auto-modifiant, durant son exécution peut lui même générer un nouveau code auto-modifiant qui sera à son tour exécuté, et ainsi de suite.

Cette notion de vague a été défini par Reynaud [59] et implémentée par Bonfante et al. [15] au sein de la plateforme CoDisasm. *The idea is to associate at any time, and to each memory address a write level and an execution level. At the beginning, for every address the execution level is set to 1 and the write level to 0. Every data written by an instruction of execution level 1 increase its write level to 1 [...] data at write level 1 is executed, thus triggering the second wave of execution, and we set the execution level to 2. In turn, wave 2 may generate a third wave and this process may repeat.*

Il est important de noter, qu'à vague donnée, le code n'est pas auto-modifiant. Cela signifie que toutes les instructions exécutées dans la vague courante étaient présentes en mémoire au moment du point d'entrée de cette même vague.

Nous utilisons dans BOA la notion de vagues définie par CoDisasm.

8.2 Objectifs

1. Dans BOA, nous souhaitons au minimum être en mesure de détecter les auto-modifications afin de ne pas effectuer un désassemblage incorrect. On entend par là le fait d'être capable de repérer si l'opcode de l'instruction que l'on s'apprête à exécuter symboliquement n'a pas été modifié en amont de l'exécution par une autre instruction.
2. Nous souhaitons être capables de continuer le désassemblage et l'exécution symbolique des différentes vagues d'auto-modification que BOA va rencontrer. L'idée étant d'obtenir le graphe de flot de contrôle de chaque vague.

8.3 État de l'art

Comme nous l'avons précédemment dit, les codes auto-modifiants et plus généralement les packers sont largement utilisés pour camoufler le code « utile » des logiciels malveillants [76]. Cette technique est efficace contre la détection des anti-virus à tel point qu'en appliquant des techniques d'apprentissage

automatique sur un grand jeu de *malware* packés, on obtient une détection des logiciels de package utilisés plutôt que des logiciels malveillants eux-mêmes [1]. Cette sur-utilisation des packers dans le domaine des logiciels malveillants conduit même la plupart des anti-virus dont la méthode de détection est basée sur une signature à déclarer un binaire « sain » comme malveillant simplement parce qu'il est packé [52].

Durant l'exécution d'un binaire packé, chaque itération de code ayant dynamiquement été générée est appelée *wave* par Reynaud [59] et Calvet et al. [19], *phases* par Dalla Preda et al. [24] ou encore *layer* par Ugarte-Pedrero et al. [71].

La grande majorité des outils de dépackage utilisent une technique d'analyse dynamique (en utilisant une machine virtuelle avec un outil d'instrumentation ou bien de l'émulation QEMU [14]) durant laquelle ils surveillent les cases mémoire modifiées puis exécutées (*"written-then-executed" instructions*) afin de détecter les différentes vagues d'exécution. On peut citer les outils suivants : PolyUnpack proposé par Royal et al. [61], Renovo de Kang, Poosankam et Yin [41], MmmBop développé par Bania [7], la plateforme SD-Dyninst de Roundy et Miller [60], CoDisasm par Bonfante et al. [15], PinDemonium de Mariani et al. [49] ou encore RePEconstruct développé par Korczynski [45].

D'autres outils suivent l'exécution à un grain différent et choisissent de baser leur analyse sur les pages mémoire. Nous pouvons citer la plateforme OmniUnpack de Martignoni, Christodorescu et Jha [50], l'outil Justin de Guo, Ferrie et Chiueh [35] ou MutantX-S proposé par Hu et al. [37].

Dans la plateforme BinUnpack, Cheng et al. [20] adoptent une approche différente en disant que le binaire a terminé sa phase de dépackage lorsque la table d'importation (IAT) a été reconstruite.

Quant à l'outil Eureka, Sharif et al. [67] utilisent une approche se basant sur les appels systèmes.

8.4 Traitement de BOA

Afin de gérer les auto-modifications ainsi que les vagues dans BOA, nous nous sommes largement inspirés du travail réalisé dans CoDisasm.

Pour les besoins de la détection des auto-modifications, nous ajoutons une liste à l'état machine s. Durant un tour de boucle de BOA, à chaque fois qu'une instruction effectue une opération d'écriture en mémoire nous allons ajouter dans cette liste l'adresse de la case mémoire où l'opération est effectuée. Ainsi, cette liste nous permet de conserver l'ensemble des cases mémoire qui ont été écrites sur un chemin d'exécution donné.

Pour une gestion correcte des vagues, en plus d'être associées à leurs adresses, les instructions sont aussi associées à leur numéro de vague. Le fait d'associer à chaque instruction son numéro de vague en plus de son adresse nous donne la possibilité dans un graphe de flot de contrôle d'avoir deux instructions différentes partageant la même adresse mais pas le même numéro de vague. Au démarrage de BOA, les premières instructions désassemblées et exécutées symboliquement appartiennent donc à la vague 0.

Pour des raisons de simplicité, nous avons omis ces détails dans le chapitre 5.

Durant le traitement d'un bloc de base par BOA, avant l'exécution symbolique de chaque instruction, BOA va vérifier si une partie ou la totalité de l'opcode de cette instruction n'est pas sur des adresses appartenant à la liste des adresses modifiées. Si c'est le cas, cela signifie que l'instruction à venir est le fruit d'une auto-modification. Il est alors nécessaire de changer de vague. À ce moment là, le numéro de vague des prochaines instructions de ce chemin sera incrémenté de 1, et la liste des adresses modifiées est remise à zéro permettant ainsi la détection possible d'une nouvelle vague.

Dans la figure 8.1, le petit code assembleur présent à gauche montre un exemple simple d'auto-modification où l'instruction `mov byte[ecx], 0x42` va modifier durant l'exécution l'opcode de l'instruction `mov eax, 0x6` présente dans la fonction `fun`. Le graphe de flot de contrôle généré par BOA présent à droite de la figure montre que l'auto-modification a remplacé dans la fonction `fun` l'instruction `mov eax, 0x6` par `mov eax, 0x42`. En regardant seulement le code assembleur, on pourrait penser au premier coup d'œil que la dernière instruction du `main` va sauter sur l'adresse `0x6`, ce qui n'est pas le cas lors de l'exécution de ce programme. Comme expliqué précédemment, les instructions dont l'adresse est préfixée par 0 appartiennent à la première vague d'auto-modification et les instructions dont l'adresse est préfixée par 1 appartiennent à la seconde vague.

```
main:
    xor eax, eax
    call fun
    mov ecx, fun+0x1
    mov byte[ecx], 0x42
    call fun
    jmp eax

fun:
    mov eax, 0x6
    ret
```

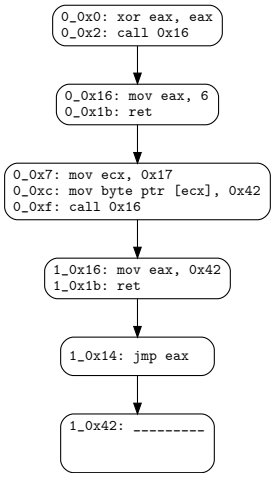


FIGURE 8.1 – Exemple simple d'un code auto-modifiant analysé par BOA

8.5 Expériences

BOA a été pensé afin de détecter la présence de code auto-modifiant dans un code binaire mais aussi afin d'être capable de continuer le déroulement de son exécution symbolique à travers les différentes vagues (comme cela est le cas lors d'une exécution réelle). Dans cette expérience, nous souhaitons en priorité savoir si BOA parvient correctement à détecter si un binaire est auto-modifiant ou non. Nous souhaitons également vérifier à quel point BOA est capable de dépaqueter des binaires protégés par des packers commerciaux et comparer les résultats de BOA aux techniques de dépaquetage « traditionnelles » faisant généralement intervenir de l'exécution dynamique.

8.5.1 Packers commerciaux Windows

8.5.1.1 Méthode

Pour cette expérience, nous utilisons le jeu de binaires *hostname.exe* packés. Nous soumettons chacun de ces binaires à BOA afin que ce dernier construise les graphes de flot de contrôle des différentes vagues qu'il aura pu traverser. Nous savons que l'ensemble des binaires du jeu utilisent des auto-modifications, nous pouvons alors compter la proportion de binaire que BOA a réussi à détecter comme auto-modifiant. De plus, comme nous connaissons le code original du binaire *hostname.exe*, nous regardons si BOA a été capable d'effectuer un dépackage complet et ainsi construire le graphe de flot de contrôle de *hostname.exe*.

8.5.1.2 Résultats

Le tableau 8.1 donne une comparaison des résultats de BOA et de 9 autres outils de dépackage. Notons que sur les 10 outils du tableau, seulement BOA aborde une approche statique alors que les 9 autres outils utilisent une analyse dynamique. 35 binaires ont été analysés par BOA. Parmi ces fichiers, BOA a réussi à dépacker complètement le code de *hostname.exe* pour 14 d'entre eux, c'est-à-dire que pour ces 14 binaires nous parvenons à récupérer toutes les vagues d'exécution. Notons également que BOA détecte la présence de code auto-modifiant dans 30 de ces binaires.

8.5.1.3 Conclusions

Bien que les résultats de BOA ne soient pas à la hauteur de ce qu'on arrive à faire avec des analyses dynamiques, cette expérience nous montre qu'une analyse statique est envisageable pour effectuer du dépackage de binaire générique, ne serait-ce que pour désassembler la première vague d'une binaire.

TABLE 8.1 – Expérience dépackage de binaires

Type d'analyse :	Statique	Dynamique								
Type de détection :		Instruction						Page mémoire	Appel système	Table d'import
Packer	**BOA**	**CoDisasm** [15]	**PinSec** [8]	**Poly-Unpack** [61]	**Renovo** [41]	**PinDemonium** [49]	**RePEconstruct** [45]	**Omni-Unpack** [50]	**Eureka** [67]	**Bin-Unpack** [20]
ACPacker	12/✓/×	635/✓/✓	-	-	-	-	-	-	-	-
ACProtect 2.0	811/✓/×	971/✓/✓	4/✓/×	-	-	-/✓	-	-/✓	-	-/✓
Armadillo 3.78	×/×	-	1/✓/×	✓/×	×/×	-	-	-/∼	-/✓	-/✓
Armadillo 9.64	×/×	165/✓/✓	-	-	-	-	-	-	-	-
ASPack 2.12	3/✓/✓	3/✓/✓	2/✓/✓	✓/∼	✓/✓	-/✓	-	-/✓	-/✓	-/✓
ASPack 2.40	3/✓/✓	3/✓/✓	-	-	-	-	-	-	-	-
EP Protector 0.3	2/✓/✓	2/✓/✓	1/✓/✓	-	-	-	-	-	-	-
eXPressor	2/✓/✓	2/✓/✓	1/✓/✓	-	-	-/✓	-	-	-	-/✓
FSG 2.0	2/✓/✓	2/✓/✓	1/✓/✓	✓/✓	✓/✓	-/✓	×/×	-/✓	-/✓	-/✓
JDPack 2.0	2/✓/×	3/✓/✓	×/×	-	-	-	-	-	-	-
MEW	2/✓/✓	2/✓/✓	1/✓/✓	✓/✓	✓/✓	-/✓	-	-/✓	-/✓	-/✓
MoleBox	2/✓/×	3/✓/✓	2/✓/✓	×/×	✓/✓	-	✓/✓	-/✓	-/✓	-/✓
Morphine 2.7	2/✓/×	3/✓/✓	-	✓/✓	✓/∼	-	-	-/✓	-/✓	-
Mystic	×/×	4/✓/✓	1/✓/✓	-	-	-	-	-	-	-
Neolite 2.0	×/×	2/✓/✓	1/✓/✓	-	-	-	✓/✓	-	-	-
nPack 1.1.300	2/✓/✓	2/✓/✓	1/✓/✓	-	-	-	-	-	-	-/✓
Obsidium 1364	6/✓/×	16/✓/✓	×/×	×/×	×/×	-/✓	-	-	-/✓	-/✓
Packman 1.0	2/✓/✓	2/✓/✓	1/✓/✓	-	-	-	✓/✓	-	-	-
PE Compact 2.20	4/✓/✓	4/✓/✓	1/✓/✓	×/×	✓/✓	-/✓	✓/✓	-	-/✓	-/✓
PE Compact 3.02.2	4/✓/✓	4/✓/✓	-	-	-	-	-	-	-	-
PELock	2/✓/×	15/✓/×	6/✓/✓	-	-	-/✓	-	-	-	-/✓
PESpin 1.1	5/✓/×	80/✓/×	×/×	-	-	-	✓/×	-	-	-/✓
Petite 2.2	2/✓/×	3/✓/✓	×/×	-	-	-	✓/✓	-	-	-/✓
RLPack	2/✓/✓	2 /✓/✓	1/✓/✓	-	-	-	-	-	-	-/✓
Setisoft 2.7.1	×/×	32/✓/×	4/✓/×	-	-	-	-	-	-	-
SVK 1.43f	2/✓/×	35/✓/×	×/×	-	-	-	-	-	-	-
tElock 0.51	5/✓/×	17/✓/✓	5/✓/×	-	-	-	✓/×	-/×	-	-/✓
tElock 0.99	14/✓/×	18 /✓/✓	-	-	-	-	-	-	-	-
Themida 1.8	3/✓/×	-	×/×	×/×	✓/∼	-	✓/×	-/✓	-/∼	-/✓
Themida 2.0.3	3/✓/×	106 /✓/✓	-	-	-	-	-	-	-	-
Upack 0.39	2/✓/×	3/✓/✓	2/✓/✓	-	-	-	-	-	-	-
UPX 2.90	2/✓/✓	2/✓/✓	1/✓/✓	✓/✓	✓/✓	-/✓	✓/✓	-/✓	-/✓	-
WinUpack	3/✓/✓	3/✓/✓	2/✓/✓	✓/∼	✓/✓	-/✓	×/×	-/✓	-/✓	-/✓
Yoda's Crypter 1.3	4/✓/✓	4/✓/✓	3/✓/×	-	-	-	-	-	-	-/✓
Yoda's Protector 1.02	2/✓/×	6/✓/✓	×/×	✓/∼	✓/∼	-	-	-	-/✓	-/✓

$n/d/r$: nombre de vagues détectées (n) / code auto-modifiant détecté (d) / récupération du code du binaire original (r)
✓ : réussite, ∼ : réussite partielle, × : échec, - : absence de résultat
La version exacte des packers utilisés est seulement connue pour BOA, CoDisasm et PinSec.

Cependant, BOA obtient de bons résultats quand il s'agit de détecter la présence de code auto-modifiant dans un binaire. Cette information peut être utile aux antivirus et nous montrons qu'il est possible de l'obtenir sans faire appel à une analyse dynamique.

8.5.2 Cheval de Troie *Emotet*

8.5.2.1 Contexte et analyse préliminaire

Emotet est un cheval de Troie bancaire observé pour la première fois en 2014. Il revient régulièrement dans de nouvelles variantes et il permet notamment de dérober des mots de passe, des listes de contacts mais aussi de se propager sur un réseau. Au second trimestre 2020, une recrudescence de campagnes d'hameçonnage visant à propager ce *malware* est observée en France [58].

Nous nous sommes procurés un échantillon de ce *malware*[6] qui semble relativement récent car le 14 Octobre 2020 il n'est détecté comme un *malware* que par 7 antivirus sur les 63 proposés par VirusTotal [72].

Nous avons dans un premier temps analysé ce binaire à l'aide d'une analyse dynamique afin de constater qu'il possède 4 vagues de code auto-modifiant.

8.5.2.2 Analyse par BOA

Nous avons soumis ce binaire à BOA afin qu'il en effectue l'analyse. De la même façon que pour toutes nos expériences BOA, seul le binaire a été donné en entrée de BOA, sans information supplémentaire.

Résultats généraux Durant son analyse, BOA parvient à analyser complètement la première vague d'exécution et à en construire le graphe de flot de contrôle. Dans cette première vague, 3341 instructions différentes sont exécutées, aucune exception n'est déclenchée et parmi les 145 instructions `RET` analysées, aucune n'est détectée comme victime d'une falsification de la pile d'appels.

Au début de l'exécution de la première vague, ce binaire utilise la fonction de l'API Windows *GetProcAddress* afin de récupérer l'adresse de différentes fonctions externes des DLL *kernel32* et *oleacc* qui ne sont pas présentes dans la table d'importation. On remarque notamment avec BOA la récupération des adresses des fonctions *InitializeCriticalSectionAndSpinCount*, *FlsAlloc*, *FlsGetValue*, *FlsSetValue*, *FlsFree*, *IsProcessorFeaturePresent*, *NotifyWinEvent*, *VirtualAllocExNuma*, *LdrFindResource_U* et *LdrAccessResource*.

Lors de l'exécution de l'instruction `0x402CC7: call ebx`, BOA détecte que la cible de ce saut dynamique est une case mémoire précédemment modifiée lors de son exécution. BOA détecte alors l'auto-modification et commence à analyser la seconde vague. L'analyse de BOA échoue durant l'analyse de la seconde vague.

Construction de la seconde vague Durant l'analyse de la première vague, après avoir exécuté environs 28 000 instructions, ce programme fait un appel à la fonction *VirtualAllocExNuma* de la DLL *kernel32* afin de demander l'allocation d'un espace mémoire d'une taille de 117 043 octets.

Quelques instructions plus loin, le binaire fait un appel à la fonction d'adresse `0x4029E0`, cette fonction contient la boucle permettant le dépackage de la seconde vague dans la zone mémoire précédemment allouée par l'appel à *VirtualAllocExNuma*. La boucle de dépackage a pour tête l'instruction d'adresse `0x402A02` et est exécutée 117 042 fois durant notre analyse.

Une fois l'exécution de la boucle de dépackage terminée, l'exécution revient dans le bloc de base d'adresse `0x402CC4` afin d'exécuter l'instruction `call ebx` d'adresse `0x402CC7` qui permet l'entrée dans la seconde vague en exécutant le code fraîchement dépacké.

Nous remarquons également qu'à chaque tour de boucle, la fonction externe *GetLastError* de l'API Windows est appelée à l'adresse `0x402A6A`. Or le résultat de cet appel, qui est normalement stocké dans le registre `EAX` n'est jamais vérifié et est écrasé par l'instruction `MOV` qui suit.

La figure 8.2 montre le bloc de base responsable de l'appel de la fonction de dépackage ainsi que le passage dans la seconde vague. La figure 8.3 montre la boucle chargée du dépackage appartenant à la fonction `0x4029E0`.

6. SHA1 : `f1ae7b90e33c979442adb82960784c3e845310cd`

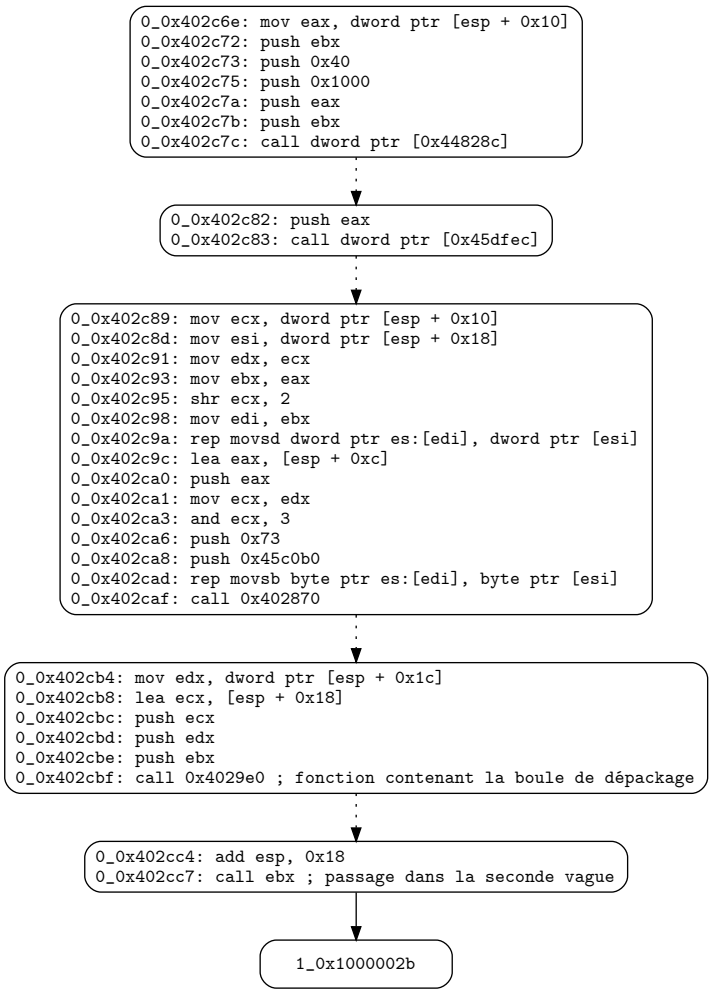


FIGURE 8.2 – Emotet : passage dans la seconde vague

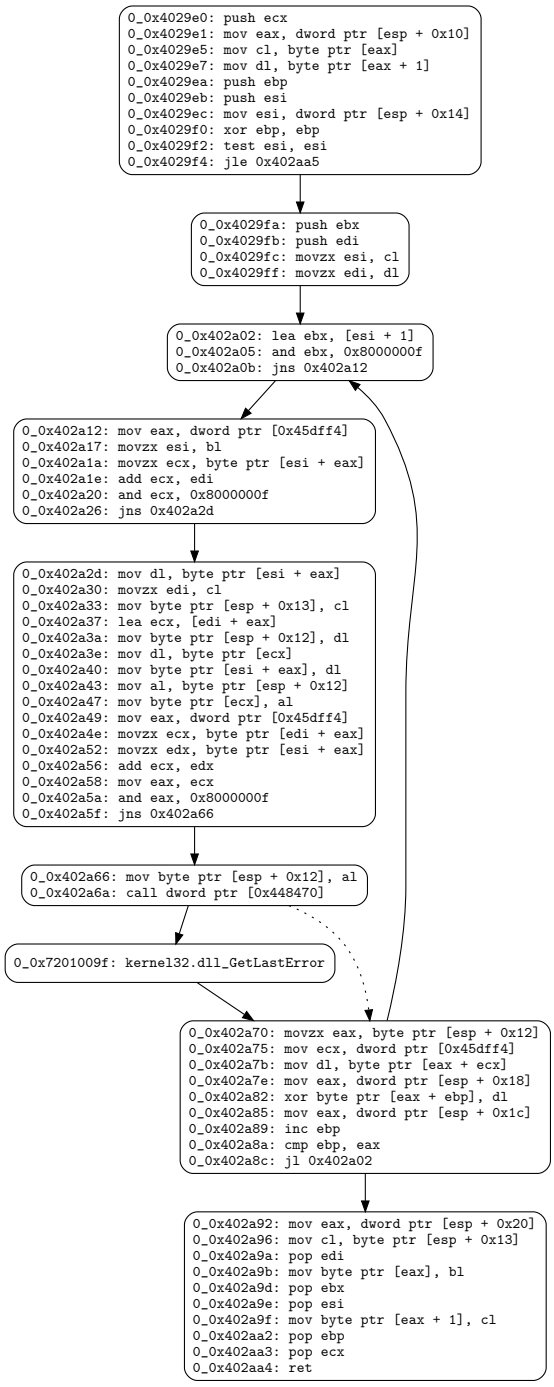


FIGURE 8.3 – Emotet : boucle de dépackage

Analyse de la seconde vague Nous avons demandé à BOA d'extraire un *dump* des octets placés dans l'espace mémoire alloué par *VirtualAllocExNuma* durant la boucle de dépackage. Nous avons soumis ce *dump* brut à VirusTotal. Bien que ce fichier ne soit pas un exécutable PE valide, 13 des 60 antivirus détectent ce fichier comme malveillant et 10 d'entre eux affirment qu'il s'agit du *malware* Emotet. Ce résultat est donné dans la figure 8.4

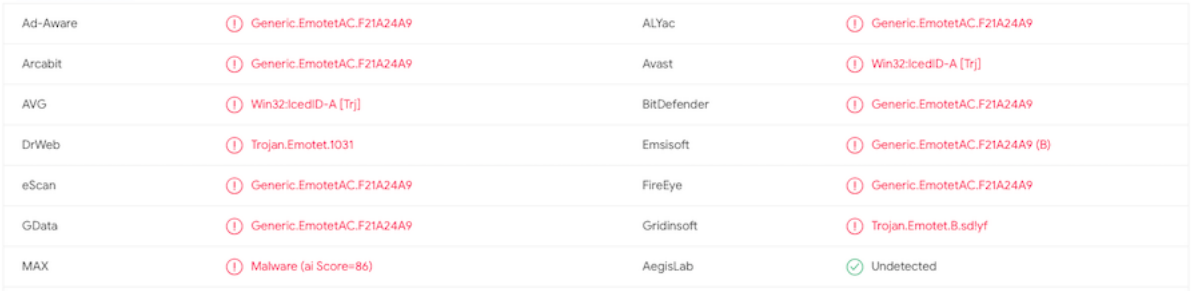


FIGURE 8.4 – Résultat VirusTotal du *dump* de la seconde vague d'Emotet

Nous avons également demandé à IDA de tenter de désassembler ce *dump* mémoire et ce dernier parvient à trouver environs 20 000 instructions différentes.

8.5.2.3 Conclusion

Cette expérience nous montre l'intérêt qu'une plateforme comme BOA pourrait avoir en amont d'une analyse antivirus classique. En effet, sans la nécessité de démarrer une machine virtuelle pour effectuer une analyse dynamique, BOA a été capable d'extraire la seconde vague de ce binaire et serait ainsi capable de la soumettre à un antivirus pour effectuer une détection à base de signature virale. De plus, obtenir des informations telles que la présence de code auto-modifiant ou de falsification de la pile d'appels permet déjà de se faire une idée sur le potentiel risque d'un binaire.

Chapitre 9

Fonctions et bibliothèques externes

L'utilisation des fonctions externes par un fichier exécutable est une pratique très courante, ne serait-ce que pour interagir avec l'API proposée par le système d'exploitation.

Ce chapitre est composé de trois sections en rapport avec la gestion des fonctions et des bibliothèques externes :

1. Dans un premier temps nous expliquons comment BOA, durant son analyse, gère les appels à des fonctions externes. Cette section correspond à l'implémentation de la partie de notre sémantique à continuation qui gère les fonctions externes (voir section 3.2.2).
2. Ensuite, dans le cadre de l'analyse d'un fichier exécutable Windows, nous expliquons comment BOA simule le chargement des DLL afin de gérer correctement les tests d'intégrité effectués par certains logiciels malveillants.
3. Enfin, nous montrons comment BOA est capable de retrouver partiellement ou complètement les entrées de la table d'importation originale d'un fichier exécutable packé.

9.1 *Hook* des fonctions externes

9.1.1 Problème

Comme cela est expliqué dans la section A.2.3.1, il est courant que durant l'exécution d'un binaire, celui-ci fasse appel à des fonctions appartenant à d'autres fichiers comme des bibliothèques externes.

Durant l'exécution symbolique d'un binaire, comme c'est le cas avec BOA, l'utilisation des fonctions externes par le binaire analysé complique notre tâche pour plusieurs raisons. Tout d'abord nous ne sommes pas forcément en possession du code des fonctions externes utilisées, il est donc difficile de continuer l'exécution symbolique normalement. De plus, même si nous avons la possibilité de suivre le flot d'exécution et d'exécuter symboliquement la fonction externe, il est parfois contre-productif de le faire. En effet, l'exécution symbolique du binaire qui nous intéresse demande déjà beaucoup de ressources et de temps, donc l'analyste n'a pas forcément envie de perdre du temps à effectuer l'exécution symbolique d'une fonction externe qui ne l'intéresse pas. D'un autre coté, l'exécution de la fonction externe va probablement modifier l'état de la machine. Ignorer son exécution peut alors engendrer des approximations et des erreurs dans l'exécution symbolique du binaire.

9.1.2 Objectif

Nous souhaitons être en mesure dans BOA de traiter correctement les appels à des fonctions externes, notamment car certains de ces appels ont une conséquence directe sur le flot d'exécution du binaire (c'est par exemple le cas avec certains packers). Cependant, pour des raisons de performance nous ne souhaitons pas exécuter symboliquement les fonctions externes appelées.

9.1.3 Choix considérés

9.1.3.1 Choix 1 : exécuter symboliquement la fonction externe

La première solution consiste à exécuter symboliquement la fonction externe appelée. L'avantage de cette solution réside dans le fait qu'elle permet de rester fidèle à une exécution réelle du programme car le flot d'exécution est respecté.

Ce choix comporte cependant plusieurs inconvénients. Tout d'abord, avant d'exécuter symboliquement la fonction externe concernée, il va être nécessaire d'accéder à son code, c'est-à-dire d'avoir accès au fichier binaire qui contient cette fonction. Le deuxième inconvénient concerne le surcoût engendré par l'exécution symbolique de la fonction externe qui peut relativement ralentir l'analyse d'un binaire.

9.1.3.2 Choix 2 : *by-passer* la fonction externe

Le choix inverse consiste à contourner l'appel de la fonction externe. C'est-à-dire que lorsque BOA détecte un `CALL` vers une fonction externe, il va en quelque sorte ignorer cet appel et continuer l'exécution symbolique du programme au niveau de l'adresse de retour du `CALL` : à l'endroit où la fonction externe aurait effectuée son retour une fois son exécution terminée. Cette solution est utilisée par David [25] (chapitre 4.6) sous le nom de *stub*.

Cette solution comporte les avantages et les inconvénients inverses de la première solution : pas de perte de temps dans l'exécution symbolique d'un code externe au programme mais une exécution symbolique moins fidèle à une exécution réelle du binaire.

9.1.4 Choix retenu dans BOA

Nous avons fait le choix dans BOA de ne pas exécuter symboliquement les fonctions externes. À la place, nous avons décidé de simuler plus ou moins l'impact engendré sur l'état de la machine par la fonction externe, notamment en fonction de la convention d'appel utilisée mais aussi de la fonction externe appelée.

Durant le déroulement de la boucle principale de BOA, si l'adresse courante à désassembler correspond au point d'entrée d'une fonction externe alors une routine particulière est appliquée afin de simuler l'impact de la fonction externe sur l'état de la machine. Les paragraphes ci-dessous décrivent le fonctionnement de la routine exécutée dans ce cas.

9.1.4.1 Convention d'appel

En fonction du type de binaire et du nom de la fonction appelée, nous pouvons savoir quelle est la convention d'appel utilisée et le nombre d'arguments attendus par la fonction. Il en existe plusieurs mais globalement nous pouvons dire qu'elles sont divisées en deux grands types : les *callee clean-up* et les *caller clean-up*. Ces deux conventions d'appel passent les paramètres via la pile, dans le premier cas cependant, c'est à la fonction appelée de « nettoyer » la pile, c'est-à-dire que c'est à elle que revient la tâche de supprimer de la pile les paramètres précédemment empilés par la fonction appelante, et ce, avant la fin de son exécution. Dans le second cas, c'est la fonction appelante qui va se charger de nettoyer la pile après l'exécution de la fonction appelée.

Dans BOA, nous simulons ce nettoyage de pile en modifiant directement la valeur du registre `ESP` dans l'environnement machine. Si la convention est *callee clean-up* alors nous augmentons la valeur de `ESP` de 4 fois le nombre d'arguments attendus par la fonction.

De plus, dans les deux cas nous devons simuler le retour de la fonction normalement effectuée par l'instruction `RET` présente à la fin de la fonction externe. Pour cela nous récupérons en haut de la pile l'adresse de retour (le *return-site*) précédemment empilé par le `CALL` appelant du programme. Cette adresse sera la prochaine adresse à être exécutée par notre exécution symbolique (l'adresse `ip`). Enfin, il reste à augmenter la valeur de `ESP` de 4 afin de simuler le *pop* effectué par le `RET`.

9.1.4.2 Modifications engendrées par la fonction externe

Outre la valeur de retour généralement renvoyée par la fonction dans le registre `EAX`, il est courant de passer en arguments de l'appel à la fonction un ou plusieurs pointeurs permettant ainsi à la fonction appelée de modifier des éléments du programme courant.

Pour gérer cette situation, nous avons défini une sémantique à continuation dans la section 3.2.2. Concrètement, un système de *hook* a été mis en place dans BOA afin de laisser le choix à l'utilisateur des modifications qu'il souhaite apporter à l'environnement machine afin de simuler l'exécution de la fonction externe. Pour cela, l'utilisateur a en sa possession le nom de la fonction et de la bibliothèque, le nombre d'arguments et quand cela est possible la valeur des différents arguments.

BOA embarque également quelques *hook* pré-définis. Par exemple, quand un programme appelle la fonction *VirtualProtect* de la bibliothèque *kernel32.dll*, BOA va inspecter les différents paramètres passés à la fonction afin de simuler le comportement de cette fonction de l'API Windows et ainsi appliquer les permissions demandées aux régions de la mémoire demandées par le programme.

9.2 Simulation de chargement des DLL (*load-time*)

Comme nous l'expliquons dans la section A.2.5.1, avant de démarrer l'exécution d'un programme, Windows va d'abord effectuer un chargement récursif des DLL dans l'espace d'adressage du processus courant. Après cela, le système d'exploitation va *patcher* chaque entrée de la table d'importation du programme avec l'adresse mémoire de la fonction externe.

Durant son exécution, BOA va simuler le comportement qu'aurait le système d'exploitation. C'est-à-dire qu'il va effectuer une recherche récursive des DLL nécessaires à l'exécution du binaire étudié. Il va charger dans l'environnement σ ces DLL, et finalement, il va modifier dans σ les différentes entrées de la table d'importation du binaire afin d'y placer les adresses des fonctions externes des DLL qu'il a placées dans σ.

9.2.1 Récupération de la liste des DLL à charger

Au début de son exécution, BOA récupère dans l'en-tête du fichier binaire à analyser la liste des DLL dont il fait appel dans sa table d'importation. Cependant, certaines DLL sont elles-mêmes dépendantes d'autres DLL. En effet, une DLL n'est rien d'autre qu'un fichier PE un peu particulier. Donc, chaque fichier DLL contient dans son en-tête sa propre table d'importation faisant ainsi potentiellement référence à d'autres DLL. De ce fait, à partir de la liste des DLL que BOA a récupérées grâce aux informations présentes dans la table d'importation du binaire, BOA va effectuer une recherche récursive à travers l'ensemble des DLL afin de récupérer la liste complète des DLL nécessaires au fonctionnement du binaire.

Pour cela, BOA est accompagné d'un dossier contenant la plupart des DLL couramment utilisées. Pour sa recherche récursive, à partir de la liste des DLL initiales, BOA va *parser* chacune d'entre-elles (à l'aide de Radare 2) afin d'en extraire la table d'importation et donc les possibles autres DLL dont elle dépend.

Un outil Windows ayant ce comportement existe déjà, il s'appelle *Dependency Walker* [26] et permet d'établir un arbre hiérarchique de l'ensemble des DLL dont dépend un fichier PE.

Finalement, après cette étape BOA est en possession de la liste complète des DLL nécessaires au fonctionnement du binaire analysé. Les DLL présentes dans cette liste sont les mêmes que celles qui seraient chargées par Windows lors d'une exécution réelle de ce binaire.

9.2.2 Chargement des DLL en mémoire

Maintenant que nous sommes en possession de la liste des DLL nécessaires, tout comme le ferait Windows, nous chargeons chaque DLL dans l'espace d'adressage du processus. Concrètement, pour chaque fichier DLL, nous copions chaque octet qu'il contient dans l'environnement mémoire σ_m. Nous commençons ce chargement à partir de l'adresse `0x72000000`.

Pour des raisons évidentes de performance, l'intégralité des fichiers DLL ne sont pas réellement copiés. Nous mettons simplement en place des marqueurs afin de savoir quelle DLL est chargée à quelle adresse

et, si durant l'exécution de BOA nous avons besoin de la valeur d'un octet présent sur une adresse où se trouve théoriquement une DLL, alors nous récupérons la valeur de cet octet à la volée dans la fichier de la DLL concernée.

9.2.3 Patch de la table d'importation

Les DLL sont maintenant chargées en mémoire, donc comme le ferait Windows, nous allons patcher la table d'importation du binaire à traiter. Nous plaçons pour chaque entrée de sa table d'importation l'adresse « réelle » de la fonction externe correspondante. Nous savons à quelle adresse de σ_m chaque fonction externe de chaque DLL se trouve en utilisant les informations présentes dans la table d'exportation des DLL.

9.2.4 Chargement des DLL à la volée (*run-time*)

Durant son exécution un programme Windows a la possibilité de charger des DLL supplémentaires, en plus de celles présentes dans sa table d'importation. Pour effectuer ce chargement « à la volée », il utilise la fonction *LoadLibrary* de l'API Windows. Cette fonction prend pour seul paramètre le nom de la DLL à charger et le système d'exploitation tente de charger cette DLL dans l'espace mémoire virtuel du processus courant.

Dans BOA, quand nous détectons durant l'exécution symbolique que le binaire fait appel à cette fonction, nous cherchons à savoir quelle DLL le binaire souhaite charger. Pour cela, nous utilisons l'état de machine afin d'essayer de récupérer la valeur du paramètre passé à cette fonction correspondant au nom de la DLL. Si nous obtenons son nom, alors de la même façon que durant le chargement du binaire, nous chargeons cette DLL dans σ_m et nous retournons (par l'intermédiaire du registre `EAX`) l'adresse mémoire à laquelle nous avons simulé le chargement de la DLL en question.

9.2.5 Discussion

Il est légitime de se poser la question de la nécessité de cette façon de faire. En effet, comme expliqué précédemment, nous n'exécutons pas symboliquement les fonctions externes mais nous nous contentons de simuler leur comportement avant de revenir sur le *return-site* de l'appelant. Il nous suffirait alors de patcher la table d'importation du binaire avec des adresses plus ou moins aléatoires et de garder de coté la valeur de ces adresses afin de savoir quand telle ou telle fonction externe est appelée.

Cependant, certains packers comme tElock 0.99 embarquent certaines fonctionnalités lui permettant de vérifier son environnement d'exécution afin de changer de comportement volontairement s'il « se croit » dans un environnement non standard. Concrètement, une des techniques qu'il utilise est la suivante. Sa table d'importation contient l'entrée d'une fonction X de la DLL *Kernel32*, lui assurant ainsi le chargement de cette DLL par le système d'exploitation avant son exécution. Durant son exécution, il va récupérer l'adresse de la fonction X dans la table d'importation et en appliquant un masque sur cette adresse il est capable de calculer l'adresse de base à laquelle la DLL *Kernel32* est chargée. À partir de cette adresse, il va lui même parser l'en-tête de *Kernel32* afin de lire sa table d'exportation. La table d'exportation d'un fichier PE contient la liste des fonctions qu'il souhaite mettre à disposition d'autres fichiers PE. Typiquement, la table d'exportation d'une DLL liste l'ensemble des fonctions qu'un programme pourra utiliser. À chaque fonction de cette liste est associée son adresse dans la *DLL*. Pour chaque fonction de *Kernel32* dont il a besoin, tElock 0.99 va vérifier si la fonction est bien présente en mémoire mais aussi si elle correspond bien a la fonction originale. En effet, nous pouvons tout à fait imaginer dans un cadre de recherche que la fonction originale de *Kernel32* soit remplacée par un *hook* utile à l'analyste. Pour effectuer cette vérification, tElock 0.99 vérifie la valeur des premiers opcodes de chaque fonction en comparant avec les valeurs qu'il attend. S'il remarque une différence alors il va faire en sorte d'avorter son exécution.

Grâce à la technique de chargement des DLL que nous utilisons, BOA est capable d'échapper à cette technique mise en place par tElock 0.99.

9.3 Récupération de la table d'importation originale d'un binaire packé

9.3.1 Problème

La table d'importation d'un binaire peut révéler des indices précieux quant à son comportement. Typiquement, si la fonction *OpenFile* de l'API Windows est présente dans la table d'importation d'un binaire, nous pouvons nous attendre à ce que ce dernier manipule des fichiers présents sur la machine.

Pour ces raisons, entre autres, les packers (mais aussi beaucoup de logiciels malveillants) masquent la table d'importation du binaire original rendant alors quasi-impossible la possibilité de connaître facilement les entrées de la table d'importation du binaire original. Afin de garantir le fonctionnement du programme initialement obfusqué, le binaire va se charger durant son exécution de reconstruire à la volée la table d'importation originale. Pour cela il utilise la fonction *LoadLibrary* de l'API Windows afin de charger les DLL nécessaires au binaire original mais aussi la fonction *GetProcAddress* qui va lui permettre de récupérer l'adresse dans différentes fonctions externes des DLL dont le binaire original a besoin.

Respectivement, ces deux fonctions nécessitent comme paramètres le nom de la DLL à charger et le nom de la fonction pour laquelle l'adresse est demandée. Bien souvent dans le cas des binaires packés, afin d'optimiser leur camouflage, les différentes chaînes de caractère utilisées lors des appels de ces fonctions ne sont pas visibles statiquement dans le binaire. Elles sont en effet déchiffrées à la volée durant l'exécution du binaire.

9.3.2 Objectif et technique utilisée

Nous souhaitons mesurer l'aptitude de BOA à détecter la phase de reconstruction de la table d'importation effectuée durant l'exécution d'un binaire préalablement protégé à l'aide d'un packer.

En réalité, BOA ne détecte pas réellement la reconstruction de la table d'importation. En fait, BOA se charge simplement d'indiquer le nom des DLL qui sont chargées durant l'exécution à la volée ainsi que le nom des fonctions externes pour lesquelles le fichier exécutable demande l'adresse. Nous supposons que ce comportement est utilisé lors de la reconstruction d'une table d'importation.

Concrètement, lors de l'analyse d'un binaire par BOA, nous gardons un œil sur les deux fonctions *LoadLibrary* et *GetProcAddress* afin de connaître les DLL et les fonctions externes récupérées par le programme, dans le plupart des cas pour la reconstruction de la table d'importation du binaire original. Pour cela nous regardons simplement le pointeur passé en paramètre de ces fonctions lors de leurs appels et nous venons lire dans l'état machine courant la chaîne de caractère pointée par ce paramètre.

9.3.3 Expérience

9.3.3.1 Méthode

Nous utilisons pour cette expérience le jeu de binaires packés précédemment utilisé dans de précédentes expériences. Nous récupérons dans un premier temps la table d'importation originale du binaire *hostname.exe*. Celle-ci contient 26 entrées correspondantes à 26 fonctions externes de 5 DLL différentes. Ensuite, pour les 35 binaires correspondants aux 35 versions de *hostname* packé, nous vérifions dans un premier temps si le binaire contient dans cette table d'importation les entrées du binaire original. Si c'est le cas, alors cela signifie que le packer en question n'effectue pas de masquage de la table d'importation du binaire qu'il protège. Dans l'autre cas, nous analysons ce binaire avec BOA et nous vérifions si nous voyons le chargement des 5 DLL nécessaires à *hostname.exe* ainsi que la récupération des adresses des 26 fonctions externes utilisées par *hostname*.

9.3.3.2 Résultats

Les résultats de cette expérience sont donnés dans le tableau 9.1. La colonne « table d'importation » indique le nombre de DLL et de fonctions externes que l'on peut statiquement observer dans la table d'importation du binaire packé. La colonne « reconstruction à la volée » correspond aux résultats obtenus durant l'analyse de BOA ; la première partie de la colonne indique le nombre de DLL chargées à la volée

TABLE 9.1 – Reconstruction de la table d'importation des binaires packés

	Table d'importation		**Reconstruction à la volée**		***hostname***	
Packer	**# DLL**	**# Fonctions**	**# DLL**	**# Fonctions**	**DLL**	**Fonctions**
ACPacker	2	5	2	28	×	×
ACProtect 2.0	2	5	3	54	✓	✓
Armadillo 3.78	3	158	0	0	×	×
Armadillo 9.64	5	232	0	0	×	×
ASPack 2.12	5	7	0	28	-	✓
ASPack 2.40	5	7	0	29	-	✓
EP Protector 0.3	5	26	0	0	-	-
eXPressor	5	16	0	26	-	✓
FSG 2.0	1	2	4	26	✓	✓
JDPack 2.0	1	5	0	0	×	×
MEW	1	2	5	26	✓	✓
MoleBox	4	37	1	108	×	×
Morphine 2.7	1	2	5	30	✓	✓
Mystic	6	268	2	0	×	×
Neolite 2.0	5	6	0	12	-	∼
nPack 1.1.300	2	5	3	28	✓	✓
Obsidium 1364	2	2	0	0	×	×
Packman 1.0	5	9	0	26	-	✓
PE Compact 2.20	1	4	4	33	✓	✓
PE Compact 3.02.2	1	4	4	33	✓	✓
PELock	1	2	0	0	×	×
PESpin 1.1	3	4	0	0	×	×
Petite 2.2	5	10	0	0	×	×
RLPack	1	5	4	26	✓	✓
Setisoft 2.7.1	2	2	0	0	×	×
SVK 1.43f	2	4	0	0	×	×
tElock 0.51	2	8	0	0	×	×
tElock 0.99	2	2	0	3	×	×
Themida 1.8	2	3	0	0	×	×
Themida 2.0.3	2	3	0	0	×	×
Upack 0.39	1	2	0	0	×	×
UPX 2.90	5	10	0	26	-	✓
WinUpack	1	2	4	26	✓	✓
Yoda's Crypter 1.3	1	2	4	36	✓	✓
Yoda's Protector 1.02	1	2	1	26	∼	×

Table d'importation : DLL/fonctions présentes dans la table d'importation du binaire
Reconstruction à la volée : DLL/fonctions chargées par le binaire au *runtime* et détectées par BOA
✓ : BOA a retrouvé toutes les DLL/fonctions initialement présentes dans l'IAT de *hostname*
∼ : BOA a retrouvé une partie des DLL/fonctions initialement présentes dans l'IAT de *hostname*
× : BOA n'a retrouvé aucune DLL/fonctions initialement présentes dans l'IAT de *hostname*
- : Les DLL/fonctions initialement présentes dans l'IAT de *hostname* sont déjà dans l'IAT de ce binaire

grâce à la fonction *LoadLibrary* alors que la deuxième partie de la colonne correspond au nombre de fonctions externes repérées durant les appels à *GetProcAddress*. Enfin, la dernière colonne représente le résultat de l'expérience et permet d'indiquer si BOA a réussi à retrouver la table d'importation originale de *hostname.exe*.

Prenons par exemple la ligne correspondant aux résultats de la version de *hostname* packé par AsPack 2.40 :

- La table d'importation de ce binaire packé contient 7 entrées correspondant à 7 fonctions externes présentes dans 5 DLL différentes (colonne « table d'importation »).
- Durant son analyse, BOA n'a remarqué aucun chargement de DLL supplémentaire (colonne « reconstruction à la volée - DLL »).
- Durant son analyse, BOA a remarqué que le binaire packé a demandé à Windows l'adresse de 29 fonctions externes différentes (colonne « reconstruction à la volée - fonctions »).
- Les 5 DLL nécessaires au fonctionnement de *hostname.exe* sont déjà présentes dans la table d'importation de ce binaire packé (colonne « hosntame » - DLL).
- BOA a repéré la récupération par le binaire packé durant son exécution des adresses des 26 fonctions externes nécessaires au fonctionnement de *hostname.exe* (colonne « hosntame » - DLL).

Sur les 35 versions protégées de *hostname.exe* nous remarquons que tous les binaires détruisent complètement ou partiellement la table d'importation du binaire original sauf le packer EP Protector. BOA parvient à retrouver complètement la table d'importation du binaire protégé pour 15 binaires et partiellement pour 2 binaires. Pour les 18 derniers cas, BOA ne parvient pas à retrouver la table d'importation de *hostname.exe*.

9.3.4 Conclusion

Nous pouvons facilement comparer ces résultats avec ceux que nous obtenons lors de l'expérience de dépackage. En effet, nous observons que la majorité des packers reconstruit la table d'importation au dernier moment juste avant de transférer l'exécution sur le code du binaire original. Ainsi, si BOA ne parvient pas à dépacker suffisamment le binaire, il n'atteint pas l'étape de reconstruction de la table d'importation et est donc incapable de retrouver les DLL et fonctions externes utilisées par le binaire original.

Rappelons cependant que l'approche de BOA est entièrement statique contrairement à des méthodes dynamiques comme celle proposée par Cheng et al. [20].

Chapitre 10

Détection et gestion des exceptions Windows

10.1 Objectifs

Nous souhaitons effectuer dans BOA une exécution symbolique la plus fidèle possible à une exécution réelle, et ce afin de s'approcher d'un graphe de flot de contrôle correct et complet.

Comme expliqué dans la section 2.2.7, les exceptions peuvent mettre à mal le désassemblage et l'exécution symbolique. Pour traiter ce problème, nous souhaitons que BOA soit capable de prendre en charge les exceptions comme le ferait le système d'exploitation lors d'une exécution réelle.

Dans un premier temps, il est nécessaire que BOA soit capable de détecter lorsqu'une exception se produit. Si BOA détecte une exception, il doit pouvoir simuler le comportement de Windows, c'est-à-dire rechercher dans la chaîne des SEH si un gestionnaire d'exception a été précédemment enregistré. Si c'est le cas, alors BOA doit effectuer la sauvegarde du contexte sur la pile et donner la main au gestionnaire d'exception. Enfin, si le gestionnaire d'exception est exécuté correctement, BOA doit être en mesure de restaurer le contexte afin de continuer l'exécution symbolique du binaire.

10.2 État de l'art

Le mécanisme de traitement des exceptions utilisé par le système d'exploitation Windows 32 bits est nommé *Structured Exception Handling* (SEH). Il existe peu de documentation sur le sujet, la meilleure explication du fonctionnement de ce mécanisme est probablement celle proposée par Pietrek [54].

Bien que plusieurs packers utilisent volontairement certaines faiblesses du SEH afin de ralentir la tâche de rétro-ingénierie, peu de littérature et d'outils traitent de ce sujet. Quand il s'agit de dépacker, la plupart des plateformes se basent sur une exécution dynamique afin de gérer les exceptions [60].

10.3 Traitement par BOA

La gestion des exceptions est traitée en plusieurs étapes par BOA. Chaque étape dépend de la précédente. Avant tout, BOA doit être en mesure de détecter si l'exécution symbolique d'une instruction déclenche une exception. Après cela, BOA recherche s'il est en mesure de calculer le point d'entrée d'un éventuel gestionnaire d'exception à exécuter. S'il en trouve un, avant de continuer l'exécution, BOA va simuler le comportement de Windows en sauvegardant le contexte sur le pile. Enfin, si l'exécution du gestionnaire d'exécution se déroule correctement, BOA va une nouvelle fois simuler le système d'exploitation en restaurant les différents registres à partir du contexte précédemment sauvegardé.

10.3.1 Détection d'une exception

Dans BOA, nous effectuons la vérification de présence d'une exception à deux moments différents : avant l'exécution d'une instruction et après l'exécution d'une instruction. Nous détectons au total 7 cas d'exceptions différents.

10.3.1.1 Instruction invalide

Lorsque BOA s'apprête à exécuter symboliquement la potentielle instruction présente à l'adresse `ip`, deux cas de figure se présentent à lui.

Si l'instruction d'adresse `ip` n'existe pas encore dans le graphe de flot de contrôle, BOA va désassembler la potentielle instruction à l'adresse `ip` dans σ_m. Si aucune instruction x86 valide n'est présente à cette adresse alors BOA lève une exception de type `ILLEGAL_INSTRUCTION`.

Cependant si l'instruction d'adresse `ip` existe déjà dans le graphe de flot de contrôle, BOA doit vérifier que cette instruction est toujours valide (cas d'auto-modification 8). Si l'opcode de l'instruction a changé, BOA désassemble la nouvelle instruction et vérifie là aussi si la nouvelle instruction à l'adresse `ip` est valide et déclenche une exception `ILLEGAL_INSTRUCTION` si besoin.

10.3.1.2 Permissions mémoire de l'instruction

Avant d'exécuter symboliquement une instruction, BOA va vérifier dans p si l'ensemble des cases mémoire sur lesquelles vivent l'opcode de l'instruction ont bien les droits en lecture et en exécution. Si ce n'est pas le cas, BOA va lever une exception de type `ACCESS_VIOLATION`.

10.3.1.3 Drapeau `TF`

Le drapeau `TF` (*Trap Flag*) correspond au bit numéro 8 du registre `EFLAGS`. Avant d'exécuter une instruction, le processeur vérifie toujours la valeur de ce drapeau. Si sa valeur est à 1 alors le processeur lève directement une exception de type `SINGLE_STEP`.

Il n'existe pas d'instruction en x86 permettant de gérer directement la valeur de ce drapeau, cependant il est assez facile de récupérer la valeur actuelle du registre `EFLAGS`, mettre à 1 le 8^e bit de cette valeur avant de la restaurer dans le registre d'état.

Comme le fait le processeur, BOA va vérifier la valeur du drapeau `TF` avant d'exécuter symboliquement une instruction. Si la valeur de ce drapeau est à 1 alors BOA déclenche une exception de type `SINGLE_STEP`.

10.3.1.4 Registres de débogage

Les registres de débogage (*debug registers*) sont généralement utilisés par les programmeurs lors du débogage de leur programme. Cependant certains packers ou logiciels malveillants en font usage afin de compliquer la tâche de l'analyste.

Ces registres sont au nombre de six. Les registres `DR0`, `DR1`, `DR2` et `DR3` contiennent les adresses pour lesquelles une exception sera levée en fonction des conditions données par la valeur du registre `DR7`. Le registre `DR7` est le registre de contrôle des registres de débogage. Il permet de contrôler la façon dont les registres de débogage vont fonctionner, c'est-à-dire quels registres de débogage doivent être activés et à quel niveau (bits 0 à 8) mais également, pour chaque registre de débogage, à quelles conditions une exception doit être levée (exécution de l'instruction, opération d'écriture, opération de lecture ou d'écriture) (bits 16 à 31). Enfin, le registre `DR6` est le registre de statut des registres de débogage, il permet d'indiquer au débogueur les conditions qui ont déclenché l'exception courante [7].

L'instruction *move to/from Special Registers* qui permet de modifier la valeur de ces registres ne peut être exécutée que par le noyau (privilège niveau 0). Cependant, il existe une technique permettant à un programme utilisateur de modifier ces registres sans avoir à utiliser cette instruction `MOV` spéciale. En effet, la valeur de ces registres est sauvegardée dans le *context record* lorsqu'une exception est levée. Il suffit alors au binaire de déclencher volontairement une exception, ensuite, depuis son gestionnaire

7. https://pdos.csail.mit.edu/6.828/2008/readings/i386/s12_02.htm

d'exception il peut modifier directement dans le *context record* les champs correspondants aux registres de débogage et enfin laisser le système d'exploitation restaurer le *context record* qu'il vient de falsifier. C'est notamment la technique qu'utilise le packer *tElock 0.99*.

Avant d'exécuter symboliquement une instruction, BOA va vérifier si l'adresse de l'instruction correspond à la valeur d'un des registres `DR0` à `DR3`. Si c'est le cas, BOA va vérifier la valeur du registre `DR7` afin de savoir si (i) le registre de débogage en question est activé et (ii) si la condition nécessaire afin de lever une exception est vérifiée. À l'heure actuelle, seulement la condition « exécution de l'instruction » est supportée par BOA. Il est cependant tout à fait possible d'ajouter le support des deux autres conditions de déclenchement.

Finalement, si la condition est valide alors BOA va déclencher une exception de type `SINGLE_STEP`.

10.3.1.5 Division

En x86, l'instruction `DIV` permet d'effectuer la division de deux nombres entiers. Par exemple avec l'instruction complète `DIV ECX`, le dividende est obtenu on concaténant les valeurs des registres `EDX` et `EAX` (div = `EDX` :: `EAX`), le quotient et le reste sont obtenus en divisant le dividende par la valeur du registre `ECX` et finalement le résultat du quotient est stocké dans le registre `EAX` et le reste dans le registre `EDX`.

Cette instruction peut déclencher une interruption dans deux cas : si le diviseur est égale à zéro ou si le résultat du quotient est trop grand pour tenir sur 32 bits. Dans les deux cas, l'exception levée est de type `INTEGER_DIVIDE_BY_ZERO`.

Quand une division est exécutée symboliquement par BOA, celui-ci va effectuer ces deux vérifications afin de déclencher ou non cette exception.

10.3.1.6 Opération en mémoire

Comme cela a été dit dans la section 3.1, chaque case mémoire se voit assigner un ensemble de permissions. Lorsque le processeur effectue une opération en mémoire, que ce soit une lecture ou une écriture, il vérifie qu'il en a le droit par rapport aux permissions associées à la case mémoire en question. Si ce n'est pas le cas, alors il lève une exception de type `ACCESS_VIOLATION`.

BOA effectue ces mêmes vérifications : pour chaque instruction qu'il exécute symboliquement il va vérifier si une lecture et/ou écriture en mémoire est effectuée. Si c'est le cas il va vérifier dans p que les opérations mémoire effectuées sont en accord avec les permissions associées à la case mémoire concernée. Si les permissions ne sont pas suffisantes alors BOA lève une exception `ACCESS_VIOLATION`.

10.3.1.7 Instruction `INT 3`

Cette instruction est habituellement utilisée par le débogueur afin de placer des points d'arrêt dans un programme en cours de développement. Elle permet de déclencher une exception de type `BREAKPOINT`, ce que BOA fait également quand il exécute symboliquement cette instruction.

10.3.2 Recherche du gestionnaire d'exception

Si BOA a déclenché une exception alors la prochaine étape est de chercher un éventuel gestionnaire pour la traiter, tout comme le ferait Windows. Pour cela, BOA va rechercher dans la chaîne des gestionnaires d'exception le dernier gestionnaire ayant été enregistré. Comme cela est expliqué dans la section A.3.2, l'adresse du premier élément de la chaîne est donnée par la valeur `FS:[0x00]` que l'on notera *SEH_chain*. Nous avons alors `@[`*SEH_chain*`]` qui correspond à l'adresse de l'enregistrement du prochain gestionnaire dans la chaîne et `@[`*SEH_chain* `+ 0x04]` qui correspond à l'adresse du point d'entrée du dernier gestionnaire d'exception enregistré dans la chaîne. Nous récupérons ces différentes valeurs dans l'environnement σ. Si une de ces valeurs est inconnue (égale à $\perp$) alors BOA interrompt l'exécution symbolique de ce chemin.

Si la valeur de `@[`*SEH_chain*`]` est égale à `0xFFFFFFFF`, cela signifie que le binaire durant son exécution n'a enregistré aucun gestionnaire d'exception qui couvre le code actuel. Durant une exécution réelle, le

gestionnaire d'exception par défaut de Windows est exécuté et il arrête le programme. Dans BOA, nous effectuons un comportement similaire en interrompant l'exécution symbolique de ce chemin.

Dans le cas contraire, cela signifie que le binaire a précédemment enregistré un gestionnaire d'exception et son point d'entrée se situe à l'adresse `@[`*SEH_chain* `+ 0x04]` que nous notons *handler_ep*.

10.3.3 Sauvegarde du contexte

Nous connaissons maintenant l'adresse du point d'entrée du gestionnaire d'exception qu'il va falloir exécuter pour gérer l'exception qui vient de se produire. Avant cela, nous devons construire les deux structures `_CONTEXT` et `_EXCEPTION_RECORD` qui sont placées sur la pile.

Concernant la structure *context record*, qui permet globalement de sauvegarder la valeur des registres CPU, elle est construite à partir des valeurs de chaque registre que l'on trouve dans σ_r à l'exception de l'entrée correspondant au registre `EIP` qui est chargée avec l'adresse de l'instruction courante (l'instruction ayant déclenchée l'exception). Une fois cette structure créée, elle est stockée sur la pile (donc dans σ_m).

Quant à la structure *exception record*, celle-ci contient plutôt des informations concernant l'exception en cours comme le code de l'exception dans le premier champ et l'adresse de l'instruction ayant causée l'exception dans le quatrième champ. Nous laissons les autres champs vides et nous stockons cette structure sur la pile (toujours dans σ_m).

Pour terminer, nous plaçons sur la pile certains pointeurs en imitant le comportement adopté lors d'une exécution réelle. La première valeur poussée sur la pile correspond à l'adresse de la structure *context record* précédemment placée sur la pile. La deuxième valeur poussée correspond à l'adresse de l'enregistrement du gestionnaire (*SEH_chain*). La troisième valeur poussée est cette fois-ci l'adresse de la structure *exception record*. Et enfin, la dernière valeur présente sur la pile correspond normalement à l'adresse de retour de la fonction dans le noyau qui est chargée de restaurer le contexte. En effet, en général à la fin de son exécution, le gestionnaire d'exception effectue un `RET` permettant ainsi de rendre la main au noyau. Comme dans notre cas nous simulons le noyau, nous plaçons sur la pile une adresse leurre ayant pour valeur `0x7CCCCCCC`. Ainsi, nous savons que si en aval une instruction `RET` effectue un saut sur cette adresse, alors c'est que nous devons simuler la restauration du contexte dans les registres.

Finalement, l'état machine est prêt pour l'exécution du gestionnaire d'exception, il ne reste plus qu'à placer dans `ip` l'adresse *handler_ep* afin de commencer son exécution.

10.3.4 Restauration du contexte

BOA exécute symboliquement le code du gestionnaire d'exception sans aucune distinction avec son comportement habituel. La seule chose que BOA va surveiller est la fameuse adresse `0x7CCCCCCC`. Si durant sa boucle d'exécution BOA doit exécuter symboliquement le bloc de base ayant pour point d'entrée l'adresse `0x7CCCCCCC` alors il sait qu'il s'agit d'un marquage particulier lui demandant de simuler la restauration du contexte suite à la fin de l'exécution d'un gestionnaire d'exception.

Pour cela BOA va effectuer l'opération inverse de la création de la structure `_CONTEXT`. Il va récupérer sur la pile le pointeur de la structure *context record* et à partir des valeurs présentes dans les différents champs de cette structure il va remplacer la valeur des différents registres dans σ_r.

Une petite exception est faite pour la valeur du champ correspondant au registre `EIP` qui ne sera pas restaurée dans un registre mais utilisée afin de mettre à jour `ip` et ainsi connaître l'adresse du prochain bloc de base à exécuter symboliquement.

10.4 Expériences

BOA permet la détection des exceptions durant son exécution symbolique, mais aussi, dans le cas d'un binaire Windows, de simuler le comportement du système d'exploitation en cherchant un potentiel gestionnaire d'exception à exécuter.

Il est courant que ce mécanisme de *try/catch* soit utilisé de façon détournée dans les packers et logiciels malveillants afin de lever volontairement une exception dans le but de compliquer la tâche d'analyse du binaire.

Nous cherchons à savoir dans cette expérience si (i) BOA est capable de détecter le déclenchement d'exception via différentes techniques et (ii) si BOA est capable de simuler correctement le système d'exploitation afin que l'exécution du binaire continue comme cela serait le cas durant une exécution réelle.

10.4.1 Méthode

Nous utilisons encore une fois pour cette expérience les binaires *hostname.exe* packés. Une petite partie des packers commerciaux que nous avons utilisés pour générer notre jeu de binaires utilisent durant leur exécution des exceptions. Lors de la génération des différentes traces d'exécution de ces binaires, nous avons pu identifier ceux utilisant des exceptions mais aussi effectuer un *dump* mémoire de chaque vague rencontrée durant l'exécution. Pour chaque vague de chaque binaire nous demandons alors à BOA d'effectuer son analyse habituelle permettant le désassemblage et la construction du graphe de flot de contrôle de la vague. Nous pouvons finalement comparer l'exécution symbolique effectuée par BOA avec l'exécution réelle dans la machine virtuelle afin de constater si BOA a réussi à détecter les exceptions mais aussi si l'exécution continue correctement après l'exception.

10.4.2 Résultats

Pour les binaires que nous avons analysés, BOA a été capable à chaque fois de détecter le déclenchement des exceptions là où elles ont eu lieu durant une exécution réelle mais aussi de simuler correctement la gestion de celle-ci afin de continuer correctement l'exécution symbolique.

Nous décrivons ici les différents cas que BOA a réussi à traiter.

10.4.2.1 tElock 0.51 : *breakpoint* `int 3` et `RET` falsifié

Le listing 10.1 montre une partie de la trace d'exécution d'un binaire packé par tElock 0.51. Les premières instructions permettent d'enregistrer dans la chaîne des SEH un gestionnaire d'exception présent à l'adresse `0x10054B3`. Ensuite le binaire déclenche volontairement une exception grâce à l'instruction `int 3` en simulant un *Software Break Point* (BPX). Enfin, depuis le gestionnaire d'exception un `RET` est utilisé afin de sauter sur une adresse calculée. Cette obfuscation est présente dans la 5^{e} vague de ce binaire.

Durant l'analyse de ce binaire, les actions de BOA sont les suivantes :

- BOA détecte qu'une exception est levée après l'exécution de l'instruction `int 3`.
- BOA calcule l'adresse du gestionnaire d'exception à exécuter en explorant la chaîne des SEH.
- BOA sauvegarde le contexte sur la pile.
- BOA calcule l'adresse de saut du `RET` falsifié et il ne restaure pas le contexte précédemment sauvegardé.

La figure 10.1 montre l'extrait du graphe de flot de contrôle généré par BOA qui concerne cette obfuscation.

10.4.2.2 tElock 0.51 : *access violation*

La listing 10.2 montre comment tElock, durant l'exécution de sa 5^{e} vague, déclenche une exception grâce à une lecture sur une case mémoire non autorisée.

BOA parvient à détecter que l'instruction `mov bx, [eax]` déclenche une exception car lors de cette exécution l'adresse pointée par le registre `EAX` ne dispose pas de la permission de lecture. Après cela BOA continue la procédure habituelle en commençant par rechercher l'adresse du gestionnaire d'exception à exécuter.

10.4.2.3 tElock 0.51 : *single step* (drapeau `TF`)

Encore une fois durant l'exécution de sa 5^{e} vague, tElock déclenche une exception de type *single step* en mettant explicitement le drapeau `TF` (*Trap Flag*) à 1. La technique qu'il utilise est donnée dans le listing 10.3.

Listing 10.1 – tElock 0.51 : *breakpoint* `int 3` et `RET` falsifié (trace d'exécution)

```
[...]
; EBP = 0x1005078
0x100548D:  push ebp ; push 0x1005078 value used later
0x100548E:  jmp 0x1005492
0x1005492:  lea eax, [ebp+0x43B] ; compute handler address
0x1005498:  jmp 0x100549B
0x100549B:  push eax ; push handler address that we want to register (0x10054B3)
0x100549C:  jmp 0x100549F
0x100549F:  xor eax, eax
0x10054A1:  jmp 0x10054A5
0x10054A5:  push dword [fs:eax] ; push previous handler record address
0x10054A8:  jmp 0x10054AB
0x10054AB:  mov [fs:eax], esp ; set our new handler record as the first chain element
0x10054AE:  jmp 0x10054B1
0x10054B1:  jmp 0x10054D1
0x10054D1:  int 3 ; trigger breakpoint exception
; exception dispatching procedure takes place here
; execute handler starting at address 0x10054B3
0x10054B3:  mov esp, [esp+0x8] ; get establisher frame address
0x10054B7:  jmp 0x10054BA
0x10054BA:  mov ebp, [esp+0x8] ; retrieve 0x1005078 value who's been pushed before
0x10054BE:  jmp 0x10054C2
0x10054C2:  lea eax, [ebp+0x462] ; compute magic value 0x10054DA from 0x1005078
0x10054C8:  jmp 0x10054CB
0x10054CB:  push eax ; push magic value
0x10054CC:  jmp 0x10054D0
0x10054D0:  ret ; use violated RET to jump on 0x10054DA computed value
0x10054DA:  [...]
```

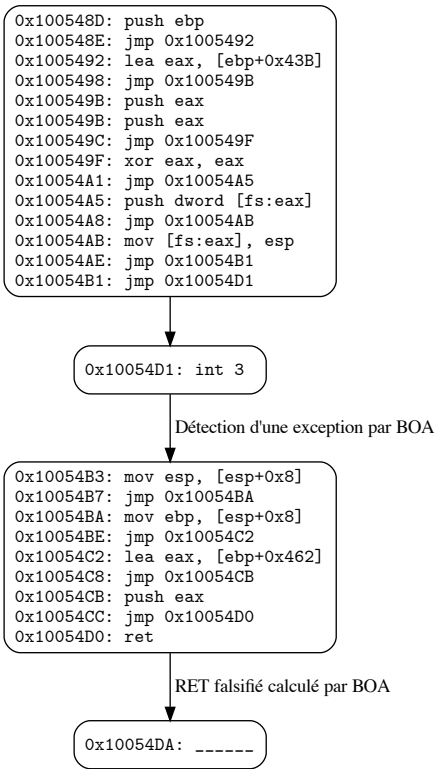


FIGURE 10.1 – tElock 0.51 : *breakpoint* `int 3` et `RET` falsifié (graphe de flot de contrôle)

Listing 10.2 – tElock 0.51 : *access violation* (trace d'exécution)

```
[...]
; EAX = 0xA00DBEEF
0x100552B:  add eax, 0x4E
0x100552E:  jmp 0x1005531
0x1005531:  mov bx, [eax] ; trigger access violation exception
; exception dispatching procedure takes place here
[...]
```

Étant donné que BOA vérifie la valeur du registre `TF` après l'exécution de chaque instruction, il est capable de détecter qu'une exception est déclenchée après l'exécution de l'instruction `popf`.

Listing 10.3 – tElock 0.51 : *single step* (trace d'exécution)

```
[...]
0x10055B4:  pushf ; push eflags register onto the stack
0x10055B5:  jmp 0x10055B8
0x10055B8:  or dword [esp], 0x100 ; Set TF flag to 1 with 0x100 mask
0x10055BF:  jmp 0x10055C2
0x10055C2:  popf ; pop stack into eflags register
; exception dispatching procedure takes place here
[...]
```

10.4.2.4 tElock 0.99 : utilisation des registres de débogage

Dans la 5e vague du binaire packé par tElock 0.99 plusieurs exceptions sont présentes. Le programme commence par déclencher une exception à l'aide de l'instruction `int 3`. Durant l'exécution de son gestionnaire il a alors la possibilité de modifier les registres de débogage afin d'enregistrer 4 adresses pour lesquelles une exception doit être déclenchée si elles sont exécutées. Plus tard, durant l'exécution, 4 exceptions sont déclenchées pour les 4 adresses précédemment enregistrées. Enfin, une dernière exception est déclenchée volontairement en effectuant une division par zéro. Un extrait de ces exceptions est donné dans le listing 10.4.

L'analyse de ce binaire avec BOA nous montre que :

- Durant l'exécution du gestionnaire d'exception déclenché par l'instruction `int 3`, BOA gère le fait que les entrées correspondantes aux différents registres de débogage sont modifiées.
- À la fin de l'exécution du gestionnaire d'exception, quand le `RET` est exécuté, BOA sait que ce `RET` n'est pas falsifié donc il simule la restauration du contexte. Durant cette procédure les registres de débogage sont restaurés avec les valeurs mises en place précédemment.
- Enfin, l'exécution du binaire continue normalement mais BOA est capable de détecter le déclenchement de 4 exceptions lorsque les 4 instructions dont les adresses sont celles des registres de débogage sont exécutées.

La figure 10.2 montre l'extrait du graphe de flot de contrôle généré par BOA qui concerne cette obfuscation.

10.4.2.5 PECompact 2.20 : *access violation* et auto-modification

Le binaire généré par le packer PECompact ne déclenche qu'une seule exception mais celle-ci est intéressante car elle combine une auto-modification et un changement de vague via la restauration de la structure *contex_record*. Concrètement, le binaire va volontairement déclencher une exception en effectuant une écriture sur une case mémoire ne possédant pas cette permission. Durant l'exécution du gestionnaire d'exception, il va modifier l'opcode de l'instruction qui a déclenché l'exception (adresse `0x10011ED`) pour y placer l'opcode d'un *jump*. Une fois l'exécution du gestionnaire d'exception terminée, Windows rend la main à l'instruction qui a déclenché l'exception (`0_0x10011ED: mov [eax], ecx`)

Listing 10.4 – tElock 0.99 : utilisation des registres de débogage (trace d'exécution)

```
[...]
; EBP = 0x10050D1
0x10050D2:  lea eax, [ebp+0x11E] ; prepare handler address
0x10050D8:  push eax ; push handler address that we want to register (0x10051EF)
0x10050D9:  xor eax, eax
0x10050DB:  push dword [fs:eax] ; push previous handler record address
0x10050DE:  mov [fs:eax], esp ; set our new handler record as the first chain element
0x10050E1:  int 3 ; trigger exception
; exception dispatching procedure takes place here
; execute handler starting at address 0x10051EF
0x10051EF:  mov eax, [esp+0x4]
0x10051F3:  mov ecx, [esp+0xC] ; get context record pointer
0x10051F7:  inc dword [ecx+0xB8] ; increment context[eip]
[...]
0x100524A:  mov eax, [ecx+0xB4] ; retrieve context[ebp] (0x10050D1)
0x1005250:  add eax, 0xCE
0x1005255:  mov [ecx+0x4], eax ; context[dr0] <-- 0x100519F
0x1005258:  mov eax, [ecx+0xB4] ; retrieve context[ebp] (0x10050D1)
0x100525E:  add eax, 0x9F
0x1005263:  mov [ecx+0x8], eax ; context[dr1] <-- 0x1005170
0x1005266:  mov eax, [ecx+0xB4] ; retrieve context[ebp] (0x10050D1)
0x100526C:  add eax, 0x70
0x1005271:  mov [ecx+0xC], eax  ; context[dr2] <-- 0x1005141
0x1005274:  mov eax, [ecx+0xB4] ; retrieve context[ebp] (0x10050D1)
0x100527A:  add eax, 0x41
0x100527F:  mov [ecx+0x10], eax ; context[dr3] <-- 0x1005112
0x1005282:  xor eax, eax
0x1005284:  and dword [ecx+0x14], 0xFFFF0FF0 ; set context[dr6]
0x100528B:  mov dword [ecx+0x18], 0x155 ; set context[dr7]
0x1005292:  ret ; gives back the hand to Windows kernel
; Windows restores context_record with EIP = 0x10050e2
[...]
0x1005112:  stc ; trigger exception because of dr3
[...]
0x1005141:  clc ; trigger exception because of dr2
[...]
0x1005170:  std ; trigger exception because of dr1
[...]
0x100519F:  cld ; trigger exception because of dr0
[...]
0x10051CE:  xor ebx, ebx
0x10051D0:  div ebx ; trigger divide by zero exception
[...]
```

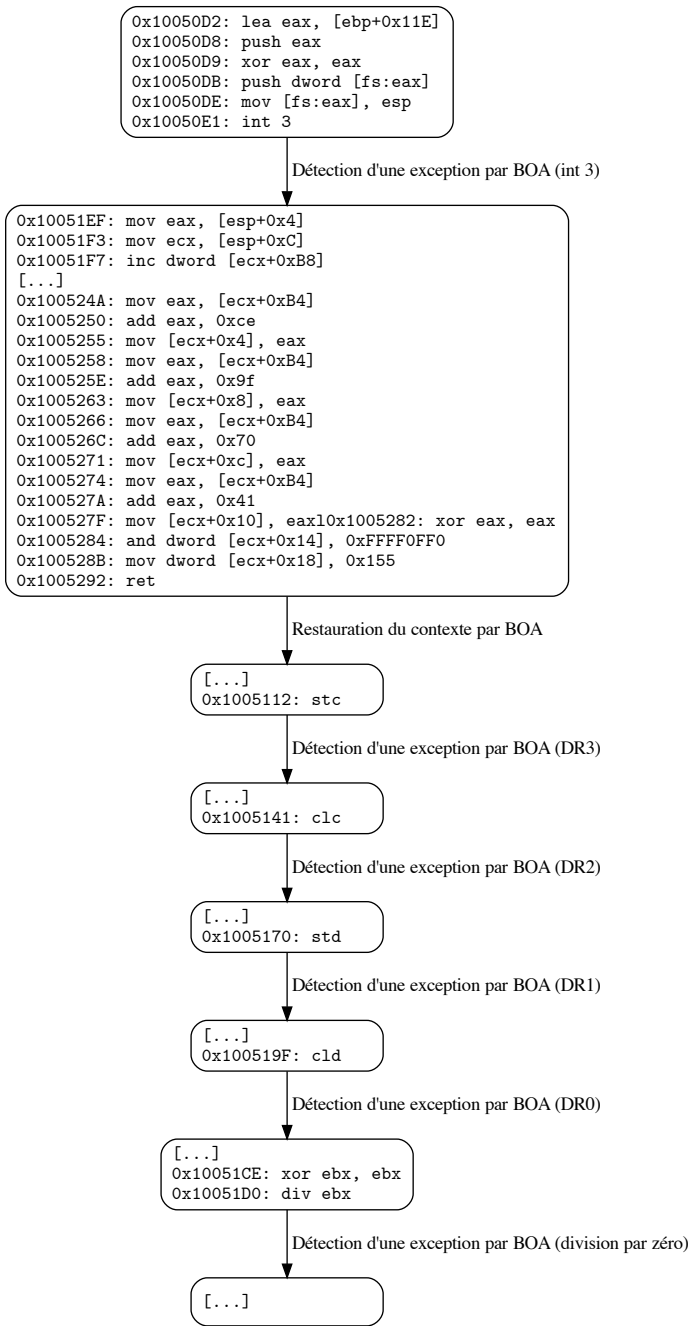


FIGURE 10.2 – tElock 0.99 : utilisation des registres de débogage (graphe de flot de contrôle)

sauf que suite à l'auto-modification l'instruction à cette adresse est maintenant un `jmp 0x1004CA7`. Le listing 10.5 montre la trace d'exécution de cette exception.

Durant l'analyse de ce binaire, BOA est capable de détecter qu'une exception est déclenchée par le `mov [eax], ecx` mais surtout, durant l'exécution du gestionnaire d'exception, BOA détecte les écritures en mémoire sur les adresses `0x10011ED` à `0x10011EF`. Après le `RET`, lorsque BOA restaure le contexte il sait que la prochaine instruction à exécuter est celle présente à l'adresse `0x10011ED`. BOA sait que les octets à cette adresse ont été modifiés, il passe alors à la vague suivante et désassemble et exécute la nouvelle instruction présente à cette adresse. La figure 10.3 montre l'extrait du graphe de flot de contrôle généré par BOA qui concerne cette obfuscation.

Listing 10.5 – PECompact 2.20 : *access violation* et auto-modification (trace d'exécution)

```
0_0x10011D7:  mov eax, 0x1004C84 ; prepare handler address
0_0x10011DC:  push eax ; push handler address that we want to register (0x1004c84)
0_0x10011DD:  push dword [fs:0x0] ; push previous handler record address
0_0x10011E4:  mov [fs:0x0], esp ; set our new handler record as the first chain element
0_0x10011EB:  xor eax, eax
0_0x10011ED:  mov [eax], ecx ; trigger access violation exception
; exception dispatching procedure takes place here
; execute handler starting at address 0x1004C84
0_0x1004C84:  mov eax, 0xF1003A09 ; compute magic value (0x1004CA7) in ecx
0_0x1004C89:  lea ecx, [eax+0x1000129E]
[...]
0_0x1004C92   mov edx, [esp+0x4] ; get exception_record address
; get ExceptionAddress field value in exception_record struct
; this is where the exception was triggered (0x10011ED)
0_0x1004C96:  mov edx, [edx+0xC];
0_0x1004C99:  mov byte [edx], 0xE9 ; self-modify first byte of 0x10011ED instruction
0_0x1004C9C:  add edx, 0x5
0_0x1004C9F:  sub ecx, edx
0_0x1004CA1:  mov [edx-0x4], ecx ; self-modify second and third byte of 0x10011ED ↩
    instruction
0_0x1004CA4:  xor eax, eax
0_0x1004CA6:  ret ; gives back the hand to Windows kernel
; Windows restore context_record with EIP = 0x10011ED
; CPU executes instruction at address 0x10011ED
; this address was self-modified by the handler
; jump in second wave
1_0x10011ED:  jmp 0x1004CA7
[...]
```

10.4.2.6 Yoda's Crypter 1.3 : transfert de l'exécution sur le binaire original

Yoda's Crypter utilise une exception afin de sauter sur le point d'entrée du binaire original. Pour cela il commence par déclencher une exception en effectuant une écriture mémoire non autorisée. Après cela, il modifie la valeur du pointeur d'instruction dans le *context_record* afin d'y placer l'adresse du point d'entrée du binaire original (dans l'exemple que nous donnons dans le listing 10.6 il s'agit du point d'entrée de `hostname.exe`).

Cette obfuscation est correctement gérée par BOA qui détecte correctement le déclenchement de l'exception ainsi que la modification de l'entrée correspondant au registre `EIP` dans le contexte.

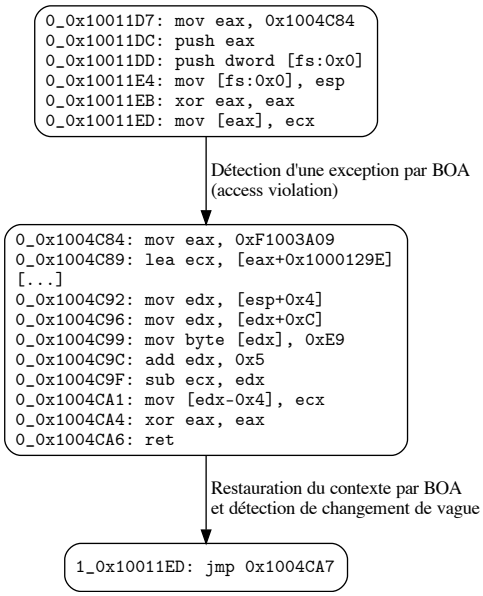


FIGURE 10.3 – PECompact 2.20 : *access violation* et auto-modification (graphe de flot de contrôle)

Listing 10.6 – Yoda's Crypter 1.3 : transfert de l'exécution sur le binaire original (trace d'exécution)

```
[...]
; EAX = 0x100590B
0x1005976:  push eax ; push handler address that we want to register (0x100590B)
0x1005977:  xor eax, eax
0x1005979:  push dword [fs:eax] ; push previous handler record address
0x100597C:  mov [fs:eax], esp ; set our new handler record as the first chain element
0x100597F:  jmp 0x1005982
0x1005982:  add [eax], al ; trigger access violation exception (EAX = 0x0 here)
; exception dispatching procedure takes place here
; execute handler starting at address 0_0100590B
0x100590B:  push ebp
0x100590C:  mov ebp, esp
0x100590E:  push edi
0x100590F:  mov eax, [ebp+0x10] ; get context_record address
0x1005913:  mov edi, [eax+0xC4] ; get ESP value from context_record
0x100591A:  push dword [edi] ; push nextSEH address on the stack
0x100591D:  xor edi, edi
0x100591F:  pop dword [fs:edi] ; set nextSEH as the new first SEH chain element
0x1005922:  add dword [eax+0xC4], 0x8 ; modify ESP field value in context_record
0x100592A:  mov edi, [eax+0xA4] ; get EBX value from context_record (0xae020023)
0x1005931:  rol edi, 0x7 ; magic computation to obtain the original binary entry point ↩
    (0x10011d7)
0x1005934:  mov [eax+0xB8], edi ; modify EIP field value in context_record with 0↩
    x10011d7
0x100593B:  mov eax, 0x0
0x1005940:  pop edi
0x1005941:  leave
0x1005942:  ret ; gives back the hand to Windows kernel
; Windows restore context_record with EIP = 0x10011D7
; CPU executes instruction at address 0x10011D7
0x10011D7:  push 0x28 ; hostname.exe entry point
[...]
```

10.4.3 Conclusion

Nous avons montré que BOA est capable de détecter correctement les exceptions mais aussi que notre approche permet de trouver, désassembler et continuer l'analyse des gestionnaires d'exception. Peu de plateformes d'analyse statique proposent la gestion des exceptions alors que cette fonctionnalité permet d'améliorer la construction du graphe de flot de contrôle mais aussi d'aiguiller le travail d'un analyste.

Chapitre 11

Conclusion

Dans cette thèse, nous avons développé une plateforme d'analyse statique nommée BOA dont l'objectif principal est la construction du graphe de flot de contrôle d'un binaire obfusqué. L'approche que nous avons choisie nous a conduits à implémenter un moteur d'exécution capable de traiter les codes auto-modifiants mais aussi à définir une sémantique à continuation permettant de simuler le système d'exploitation dans le cadre des exceptions et des appels système. Afin de calculer les adresses de saut des instructions dynamiques, notre outil calcule les états partiels de la machine en utilisant une exécution symbolique couplée à une analyse de teinte. Nous avons validé expérimentalement cette plateforme d'analyse sur un jeu de binaires obfusqués à l'aide de Tigress, sur un jeu de binaires packés ainsi que sur les deux logiciels malveillants XTunnel et Emotet.

Par ce travail, nous avons montré qu'une approche basée sur l'analyse statique permet d'obtenir des résultats convaincants face à des obfuscations jusque-là majoritairement traitées grâce à des analyses de type dynamique. Nous avons par exemple montré que notre approche, bien que statique, est capable d'effectuer le dépackage complet de binaires protégés par différents packers Windows commerciaux. Nous pensons que cette fonctionnalité permettant de dépacker partiellement ou complètement un code protégé pourrait tout à fait trouver sa place en amont d'un antivirus classique afin d'améliorer sa détection. Nous avons également pu voir que BOA est capable de détecter le déclenchement de différents types d'exceptions mais surtout de simuler le comportement de Windows permettant ainsi de continuer correctement l'exécution symbolique d'un binaire après le déclenchement d'une exception. Le déclenchement volontaire d'une exception est une technique d'anti-analyse très facile à mettre en œuvre. Pour autant, et contrairement aux analyses dynamiques, la majorité des outils d'analyse statique actuels ne permettent pas de détecter et gérer ce genre d'obfuscation.

La principale limite de notre approche réside dans l'utilisation d'un solveur SMT qui se trouve être le goulot d'étranglement de la plateforme. Le temps d'analyse nécessaire à BOA pour traiter un fichier exécutable est directement lié au temps de calcul nécessaire au solveur SMT pour exécuter symboliquement chaque bloc de base. Cette faiblesse se fait vraiment sentir quand il s'agit de traiter des binaires contenant beaucoup de codes exécutables. Afin de résoudre en partie ce problème, il serait nécessaire d'effectuer un travail d'ingénierie au sein de la plateforme BOA dans le but de tirer partie des différentes techniques de *multithreading* existantes et ainsi explorer plusieurs chemins d'exécutions simultanément.

Comme nous l'avons expliqué, les logiciels font régulièrement appels aux API proposées par le système d'exploitation hôte. L'impact de ces appels sur l'état de la machine peut énormément influencer le chemin d'exécution que va prendre un programme durant son exécution. Bien que BOA intègre des *hooks* pour la plupart des API courantes et simule au mieux leur comportement, il serait nécessaire d'améliorer cette fonctionnalité de BOA en supportant plus de fonctions d'API.

Nous notons également que de plus en plus de logiciels malveillants utilisent plusieurs *thread* et/ou plusieurs processus durant leur exécution. Or, à l'heure actuelle, l'exécution symbolique de BOA n'est capable de gérer qu'un seul processus utilisant qu'un seul *thread*. Il serait nécessaire d'ajouter à BOA ce support afin de pouvoir traiter les binaires utilisant ces techniques.

Dans cette thèse, nous avons surtout évalué l'approche de BOA seul, en tant qu'outil d'analyse de binaire. Il serait intéressant de mesurer l'impact de l'utilisation de BOA au sein d'une chaîne d'outils,

notamment antivirus, et ainsi évaluer en quoi la présence de BOA améliore ou non la détection des logiciels malveillants.

Annexe A

Prérequis : de la machine au programme

A.1 Machine x86 : description et fonctionnement

A.1.1 Description générale

Nous désignons ici par le mot « machine » un ordinateur composé d'un processeur et d'une mémoire. Le processeur (souvent noté CPU pour *Central Processing Unit*) est l'organe chargé d'exécuter les instructions machine des programmes informatiques. Nous pouvons voir une instruction machine comme une opération arithmétique ou logique que le processeur va savoir traiter. Le processeur embarque généralement un nombre fini d'emplacements dans lesquels il peut stocker temporairement des informations comme par exemple le résultat d'un calcul. Ces emplacements sont les registres CPU, ils sont identifiés de façon unique par leur nom. Les informations peuvent également vivre dans la mémoire qui peut être vue comme un tableau unidimensionnel. On peut ainsi comparer un élément du tableau identifié par son indice par une case mémoire identifiée par son adresse. Le processeur peut ainsi récupérer (lecture) ou sauvegarder (écriture) des informations dans la mémoire sur une ou plusieurs cases. Les composants assurant le rôle de mémoire sont généralement de deux types différents : une mémoire est dite « morte » si les données sont persistantes après la mise hors tension de la mémoire, les disques durs ou encore les clés USB sont des mémoires mortes, dans le cas contraire la mémoire est dite « vive », c'est le cas des barrettes de RAM.

A.1.2 Processeur et architecture

Outre les différentes caractéristiques techniques qui définissent les processeurs (comme la cadence de son horloge ou encore le nombre de cœurs de calcul qu'il embarque), les processeurs sont classés en différentes familles que l'on appelle architectures.

L'architecture d'un processeur correspond à sa spécification fonctionnelle du point de vue du programmeur en langage machine. Elle définit notamment son jeu d'instruction, ses registres (nombre, taille et noms), la taille maximale des mots que le processeur sait manipuler ou encore la spécification des entrées-sorties et de l'accès à la mémoire.

Les plus connues sont sans doute l'architecture x86 (appartenant à la famille des architectures CISC) utilisée par les processeurs Intel et AMD que l'on trouve habituellement dans les ordinateurs de bureau et les ordinateurs portables, et, l'architecture ARM (appartenant à la famille des architectures RISC) que l'on trouve maintenant dans la quasi totalité des smartphones, tablettes et autres objets connectés. Notons que l'architecture ARM gagne petit à petit du terrain sur l'architecture x86. De plus en plus de constructeurs l'adoptent dans leurs ordinateurs comme Apple qui propose maintenant à la vante des machines ARM.

A.1.2.1 Registres CPU

Les processeurs d'architectures différentes embarquent des registres différents tant par leurs noms, leurs nombres et leurs utilisations. Nous prenons ici le cas d'une machine x86-32 bits ; typiquement cette machine embarque les registres de taille 32 bits suivant :

- Les registres d'usage général : `EAX`, `EBX`, `ECX` et `EDX`.
- Les registres utilisés pour la pile : le pointeur de haut de pile `ESP` et le pointeur de base de pile `EBP`.
- Les registres `ESI` et `EDI` typiquement utilisés pour parcourir des tableaux
- Les registres de segment `CS`, `DS`, `SS`, `ES`, `FS` et `GS` utilisés pour former une adresse mémoire.
- Le registre d'état `EFLAGS` utilisé afin de donner des informations sur l'exécution de la dernière instruction. Par exemple le 6^{e} bit, nommé `ZF` est positionné à 1 si le résultat de la dernière instruction était nul et à 0 sinon. Ce registre est surtout utilisé par les instructions de branchement.
- Le registre `EIP` (aussi appelé registre de pointeur d'instruction) qui contient l'adresse mémoire de la prochaine instruction à être exécutée par le CPU.

A.1.2.2 Instruction machine

Une instruction, ou instruction machine, est une opération élémentaire que le processeur sait effectuer, c'est l'ordre le plus basique que l'ordinateur peut comprendre.

Tous les processeurs d'une même architecture sont capables de comprendre un ensemble fini d'instructions, on appelle cet ensemble le jeu d'instructions du processeur. Par exemple, le jeu d'instruction x86 est donné par la documentation Intel [38].

Les instructions sont codées en binaire, et le code binaire d'une instruction (souvent représenté sous sa forme hexadécimale) est appelé l'opcode. C'est l'opcode d'une instruction qui dicte au processeur l'opération à effectuer.

La taille d'une instruction est donnée par le nombre d'octets que comporte son opcode.

Afin de faciliter la lecture des instructions pour les humains, nous utilisons généralement le langage assembleur utilisé par le processeur. Il permet de traduire l'opcode d'une instruction (binaire) en des symboles appelés mnémoniques dont le sens est plus évident pour les humains.

Grâce au langage assembleur, il est possible de traduire le mnémonique d'une instruction en son opcode et inversement.

Par exemple, l'instruction x86-32 de mnémonique `MOV EAX, EBX` a pour opcode `89D8` et est de taille 2. Cette instruction demande au processeur de copier la valeur contenue par le registre `EBX` dans le registre `EAX`.

Les instructions (sous-entendu les opcodes des instructions) vivent dans la mémoire de la machine. L'opcode d'une instruction peut être composé d'un ou plusieurs octets. Étant donné qu'une case mémoire ne peut contenir qu'un seul octet, l'opcode d'une instruction peut vivre sur plusieurs cases contiguës de la mémoire.

Types d'instruction Les différentes opérations effectuées par une instruction peuvent être séparées en deux catégories. La première catégorie contient les opérations dites globales, ce sont les actions « effectives » directement liées au mnémonique de l'instruction que l'on s'attend à voir. Par exemple, pour l'instruction `MOV EAX, EBX`, la seule action globale est la copie du contenu du registre `EBX` dans le registre `EAX`. La deuxième catégorie contient les actions dites logistiques, elles permettent une fois les actions globales réalisées par le CPU, d'effectuer les changements nécessaires afin, par exemple, de préparer le CPU à l'exécution d'une nouvelle instruction.

Une des actions logistiques commune à beaucoup d'instructions concerne le chargement du registre pointeur d'instruction (le registre `EIP` dans une machine x86-32 bits). La valeur contenue dans ce registre permet d'indiquer au processeur à quelle adresse en mémoire se trouve le premier octet de l'opcode de l'instruction qu'il doit exécuter. En d'autres termes, cette action permet à l'instruction courante de dicter au CPU quelle sera la prochaine instruction à exécuter.

En fonction du type de modification qui est engendré sur le registre pointeur d'instruction (souvent noté `ip` pour *Instruction Pointer*), nous catégorisons différents types d'instructions (les exemples donnés

utilisent le jeu d'instructions x86-32 bits) :

- **Séquentielle :** Instructions pour lesquelles `EIP` est chargé avec l'adresse de la prochaine instruction en mémoire (ex. `MOV AL, 61h`).
- **Saut inconditionnel explicite :** Instructions pour lesquelles `EIP` est chargé avec l'adresse explicitement donnée par l'instruction (ex. `JMP 0x1005121`).
- **Saut conditionnel :** Instructions pour lesquelles la valeur chargée dans `EIP` est fonction de la condition attendue par l'instruction et de la valeur du registre d'état (ex. `JNZ 0x1005324`) :
  - Si la condition est évaluée à *False* alors `EIP` est chargé avec l'adresse de la prochaine instruction en mémoire,
  - mais si la condition est évaluée à *True* alors `EIP` est chargé avec l'adresse explicitement donnée par l'instruction.
- **Saut dynamique :** Instructions pour lesquelles `EIP` est chargé avec une valeur dépendant de l'état actuel de la machine, cette catégorie peut être divisée dans les sous-catégories suivantes :
  - **Les `JMP` dynamiques :** `JMP EAX`, `JMP [0x12345678]`, etc.,
  - **Les `CALL` dynamiques :** `CALL EAX`, `CALL [EAX]`, etc.,
  - **Les `RET` :** l'instruction *Return from Procedure*,
  - **Les `INT` :** l'instruction d'interruption logicielle.

A.1.2.3 Décodage d'une instruction x86

Les instructions de l'architecture x86 ne sont pas toutes de taille identique. Les plus petites sont composées d'un opcode d'un octet seulement. Intel assure cependant qu'une instruction valide ne peut pas dépasser la taille de 15 octets.

Lorsque le processeur s'apprête à exécuter une instruction, il ne connaît au départ que l'adresse mémoire du premier octet de son opcode. Les octets de l'opcode d'une même instruction étant contigus en mémoire, le processeur va lire un à un les octets dans la mémoire à partir de l'adresse `ip` présente dans le registre pointeur d'instruction `EIP` jusqu'à ce que la suite des octets lus forme l'opcode d'une instruction valide dans le jeu d'instructions du processeur.

Cette instruction, notée `i` a pour adresse `ip`, l'adresse mémoire du premier octet de l'opcode de `i`. Ce mécanisme est décrit par la fonction `DecodeInstr` de l'algorithme 8 qui décode une instruction en mémoire à partir de l'adresse a.

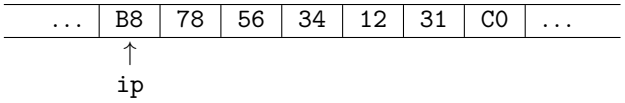
Algorithme 8 : Décodage d'une instruction x86

```
Function DecodeInstr(ip, memory) :
    Input : ip, l'adresse de l'instruction à décoder et memory, la mémoire de la machine
    Output : opcode, le code machine de l'instruction x86 valide à l'adresse ip
    opcode = ∅
    foreach j ∈ [0, 15] do
        opcode = opcode :: memory[ip + j]
        /* Vérifie si l'opcode courant correspond à une instruction x86 valide */
        if IsValidInstr(opcode) then
            return opcode
```

Une fois l'instruction décodée par le processeur, ce dernier va traiter l'opération dictée par l'opcode de `i`, on dit que le processeur exécute `i`.

Finalement, en fonction du type de `i`, une nouvelle adresse est chargée dans le registre `EIP` permettant au processeur de décoder et exécuter une nouvelle instruction.

Exemple Supposons que le registre `EIP` vient d'être chargé avec l'adresse `ip` et que le contenu de la mémoire à l'adresse `ip` soit :



Le processeur va décoder l'opcode `B878563412` qui correspond à l'instruction `MOV EAX, 0x12345678`. Cette instruction est exécutée par le CPU, conduisant le registre `EAX` à être chargé avec la valeur `0x12345678`. Cette instruction étant séquentielle, le pointeur d'instruction va être chargé avec la valeur $\texttt{ip}' = \texttt{ip} + 5$, 5 étant la taille de la dernière instruction exécutée.

A.1.3 Mémoire

Dans le cas d'une machine x86-32 bits, la mémoire est un tableau contenant 2^{32} cellules pouvant chacune stocker une valeur de la taille d'un octet (un nombre dont la valeur peut être comprise entre 0 et 255). Chaque case mémoire est identifiée par son adresse, un nombre dont la valeur est comprise entre 0 et 2^{32} - 1.

A.1.4 Pile

Les processeurs et instructions machine utilisent régulièrement des structures de données sous la forme d'une pile. Lors de l'exécution d'un programme, la pile sert habituellement à stocker des variables temporaires ou encore des paramètres lors de l'appel d'une fonction. Les machines x86 ne proposent pas de piles au niveau matériel. Pour l'implémentation de cette structure de données, on utilise à la place une portion contiguë de la mémoire en combinaison des deux registres `ESP` et `EBP` qui servent respectivement de pointeurs de haut de pile et de base de pile. De plus, certaines instructions machine permettent d'interagir directement avec la pile comme par exemple l'instruction `PUSH EAX` permettant d'empiler la valeur présente dans le registre `EAX` en haut de pile.

A.2 Systèmes d'exploitation et fichiers exécutables

A.2.1 Introduction sur les systèmes d'exploitation

Un système d'exploitation (souvent noté OS pour *Operating System*) est un logiciel. Lors de la mise en marche de la machine, il est un des premiers programmes à être exécuté et généralement un des derniers à s'arrêter lorsque l'on éteint la machine.

Ce logiciel principal a énormément de rôles, il propose une interface utilisateur, il est responsable de la gestion des différentes ressources (mémoire, processeur, carte graphique, ...), il permet de démarrer d'autres logiciels comme un navigateur internet ou un logiciel de traitement de texte, il assure la gestion des utilisateurs ou encore l'organisation et la manipulation des fichiers. Cette liste reste cependant non exhaustive.

Il existe des dizaines de systèmes d'exploitation dont le code source est libre ou non, nous pouvons citer Windows, macOS, GNU/Linux ou encore Android.

A.2.1.1 Interface de programmation

Le système d'exploitation se charge de faire l'intermédiaire entre les ressources matérielles et les applications utilisateur ; ainsi les applications n'ont pas accès directement aux ressources matérielles et doivent nécessairement passer par le système d'exploitation pour, par exemple envoyer un paquet sur le réseau.

Pour permettre cela, les systèmes d'exploitation proposent généralement une interface de programmation (communément appelée API pour *Application Programming Interface*). Cette API, propre à chaque système d'exploitation, offre un ensemble normalisé de fonctions que les applications peuvent utiliser afin d'accéder aux différentes ressources matérielles par exemple.

Dans le cas de Windows, cette API s'appelle Win32, elle est proposée par l'intermédiaire de trois bibliothèques logicielles : *Kernel32.dll*, *User32.dll* et *GDI32.dll*.

A.2.2 Utilisation du mot « programme »

D'après Wikipédia, « un programme informatique est un ensemble d'opérations destinées à être exécutées par un ordinateur » [56]. Un programme (sous-entendu informatique) peut être de différentes formes, par exemple un programme source est un code écrit dans un langage de programmation comme le C++ (habituellement par un humain) et est destiné à être compilé. À ce niveau là, il n'est pas encore compréhensible par le processeur d'une machine.

Un programme binaire contient quant à lui les instructions qu'un processeur pourra exécuter, c'est la forme que l'on obtient après la compilation d'un programme source.

Le mot « programme » pouvant désigner à la fois un fichier source ou un fichier binaire, nous utilisons la désignation « fichier exécutable » ou encore « binaire » pour parler d'un programme binaire exécutable.

A.2.3 Notion de compilation

L'opération de compilation consiste à transformer un code source (par exemple un programme écrit en C++) en du code dit objet, généralement en langage machine. Le code généré est alors compréhensible par le processeur qui peut l'exécuter.

A.2.3.1 Les fonctions et bibliothèques externes

Les développeurs informatiques ont pour principe de base de ne pas « réinventer la roue ». Dans notre cadre, il faut comprendre qu'il est en général inutile de programmer à nouveau des fonctions qui sont susceptibles de déjà exister quelque part comme par exemple la fonction `random`. Pour cela, lors du développement d'un logiciel, il existe des mécanismes permettant d'utiliser des fonctions déjà existantes dans d'autres programmes. Ces programmes sont généralement appelés bibliothèques logicielles. Par exemple, c'est par ce biais que Windows met à disposition des programmeurs, son API. Il propose l'utilisation de plusieurs bibliothèques logicielles (appelées DLL pour *Dynamic Link Library*) dans lesquelles sont disponibles de multiples fonctions permettant par exemple l'accès à un fichier du disque dur.

A.2.3.2 Compilation statique ou dynamique ?

Lorsqu'un développeur conçoit un logiciel utilisant une ou plusieurs fonctions d'une bibliothèque externe, deux choix s'offrent à lui : la compilation statique ou la compilation dynamique.

Dans le premier cas, lorsque le compilateur va générer le fichier exécutable, il prendra le soin d'embarquer dans le binaire final l'ensemble du code des bibliothèques externes utilisé par le logiciel. De ce fait, lors de son exécution, l'ensemble des fonctions nécessaires au fonctionnement du binaire seront déjà présentes dans celui-ci et aucune étape supplémentaire ne sera nécessaire pour permettre son fonctionnement.

Dans le cas d'une compilation dynamique, le compilateur va simplement insérer à un endroit particulier du binaire une liste contenant l'ensemble des noms des fonctions et l'ensemble des noms des bibliothèques externes utilisés dans le logiciel. Ce sera seulement au moment de l'exécution du binaire, que le système d'exploitation devra faire le nécessaire afin de (i) vérifier qu'il est en possession des bibliothèques nécessaires au logiciel et (ii) charger en mémoire ces bibliothèques. Le système d'exploitation s'occupe également d'indiquer au binaire (via l'intermédiaire de sa table d'importation) à quelle adresse mémoire se trouve chacune des fonctions externes dont il a besoin. Dans les détails, cette étape diffère d'un système d'exploitation à l'autre.

Ces deux méthodes ont leurs avantages et inconvénients : une compilation statique générera des binaires plus lourds car ils devront contenir tout le code nécessaire provenant des bibliothèques externes alors qu'un logiciel compilé de façon dynamique pourra fonctionner sur une machine ne disposant pas de la bibliothèque externe.

La compilation dynamique est bien souvent la méthode choisie par défaut par les compilateurs. Par exemple, l'option `-static` doit être spécifiée au compilateur GCC afin qu'il effectue une compilation statique.

A.2.4 Structure d'un fichier exécutable

Les programmes, qu'ils soient compilés pour les systèmes d'exploitation Linux, Windows ou macOS, sont plus ou moins structurés de la même façon, ils contiennent différentes informations et sont globalement organisés en sections.

En tête des sections, les binaires disposent d'une en-tête qui va contenir les informations dont le système d'exploitation a besoin pour mettre en mémoire le programme. Parmi les informations que le *header* nous apporte, nous retrouvons pour chaque section, leur taille ainsi que l'offset dans le fichier où se situe leur premier octet. De plus, nous trouvons le point d'entrée du programme qui indique au CPU la localisation dans la section `text` du premier octet à exécuter.

Les instructions du programme sont généralement situées dans la section `text`. Cependant, il est important de noter que cette section ne contient pas nécessairement que du code exécutable mais aussi toutes sortes de données qui peuvent être mélangées au milieu du code. D'autre part, du code peut être stocké dans d'autres sections.

A.2.4.1 Quelques éléments importants dans un binaire exécutable

Type Il existe différents formats pour les fichiers exécutables, par exemple le format PE pour Microsoft Windows, ELF pour Linux ou encore Mach-O pour macOS. Lorsque l'on analyse un binaire, la connaissance du format est utile pour plusieurs points. Par exemple, le calcul de la correspondance « adresse virtuelle / *raw offset* » n'est pas le même entre les fichiers PE et ELF. Ou encore, l'espace d'adressage virtuel du processus qui est organisé différemment entre Windows et Linux.

Segments et sections Les fichiers exécutables sont généralement divisés en différents segments et sections (à noter que la notion de segment n'existe pas pour les fichiers PE). Chaque segment/section est caractérisé par son nom, son adresse virtuelle, sa taille, son offset et les différentes permissions d'accès (lecture, écriture, et exécution) qui lui sont associées. Ces informations nous permettent par exemple de pouvoir calculer l'offset dans le fichier binaire d'une adresse virtuelle ou encore de vérifier qu'une opération en mémoire est valide au regard des permissions de la case mémoire concernée.

Les symboles importés Dans le cas d'une compilation dynamique, le nom des fonctions ainsi que le nom des bibliothèques sont stockés dans une ou plusieurs structures de l'en-tête du binaire. Le nom et le fonctionnement de ces structures diffèrent d'un système d'exploitation à l'autre, dans le cas d'un fichier PE, ces informations sont stockées dans la table d'importation (souvent notée IAT pour *Import Address Table*).

Les différentes entrées de la table d'importation permettent au système d'exploitation de savoir quelles bibliothèques externes il doit charger dans l'espace mémoire du processus permettant l'exécution du binaire. De plus, un système de mappage permet au système d'exploitation de retrouver l'adresse virtuelle de l'entrée d'une fonction externe lorsque celle-ci est appelée durant l'exécution du fichier exécutable.

Autres informations D'autres informations utiles peuvent être récupérées dans l'en-tête du binaire comme l'adresse du point d'entrée du binaire ainsi que l'adresse de base.

A.2.5 Exécution d'un fichier exécutable

Généralement les programmes exécutables sont stockés sous la forme d'une fichier (par exemple un fichier *exe* dans le cas de Windows) sur un support tel qu'un disque dur.

Afin de pouvoir exécuter un binaire, le système d'exploitation doit d'abord « charger » le fichier en mémoire (vive). Pour cela, un processus dédié à l'exécution du fichier est créé. Celui-ci dispose de son propre espace d'adressage virtuel.

Lors de la création du processus, une des étapes consiste à copier les octets de chaque section du programme à l'adresse correspondante en mémoire.

Une fois cette étape réalisée, l'adresse du point d'entrée du binaire est chargée dans le registre pointeur d'instruction permettant au processeur de savoir à partir de quelle adresse mémoire il doit décoder l'opcode de la première instruction du programme à exécuter.

A.2.5.1 Chargement des bibliothèques externes (Windows)

Avant d'exécuter la première instruction du binaire, le système d'exploitation va d'abord s'occuper de charger les bibliothèques dont le logiciel a besoin pour fonctionner correctement.

Ce mécanisme est différent d'un système d'exploitation à l'autre. Nous prenons ici l'exemple de Windows car notre travail porte principalement sur des binaire de type PE.

Pour effectuer cette étape, Windows va parcourir la table d'importation du binaire afin d'obtenir la liste des DLL nécessaires. Il va alors charger en mémoire (dans l'espace d'adressage du logiciel à exécuter) chacune d'entre elles. Il est important de noter qu'une DLL étant elle-même un fichier PE, elle peut elle aussi nécessiter le chargement d'autres DLL dont elle dépend ; cette opération est donc récursive.

En plus de contenir la liste des bibliothèques externes nécessaires, la table d'importation agit comme un tableau de correspondance dans lequel on associe à chaque nom de fonction externe l'adresse mémoire à laquelle son point d'entrée se trouve. Comme l'adresse mémoire à laquelle les DLL sont chargées peut varier d'une exécution à l'autre, il est nécessaire pour le système d'exploitation de patcher cette table de correspondance à chaque exécution. On entend par là le fait de remplacer dans la table d'importation les adresses des différentes fonctions fraîchement chargées en mémoire afin que l'exécution du binaire se déroule correctement.

A.3 Interruptions et exceptions

En informatique, les interruptions et les exceptions sont des éventements inattendus qui perturbent le déroulement normal de l'exécution effectuée par le processeur.

Une interruption est un signal envoyé par un périphérique au processeur afin de le prévenir qu'un événement requiert une attention immédiate. Par exemple, l'appui sur une touche du clavier produit une interruption, le processeur arrête alors temporairement ce qu'il est en train de faire afin de traiter l'interruption qu'il reçoit. Bien qu'une interruption soit généralement déclenchée par un élément matériel, il est également possible de déclencher une interruption de façon logicielle grâce à l'instruction `INT n`. Cette instruction est généralement utilisée pour implémenter les appels systèmes ou bien pour notifier un débogueur d'un événement spécifique.

Contrairement aux interruptions qui correspondent généralement à des événements extérieurs au processeur, les exceptions sont déclenchées quand une erreur apparaît lors de l'exécution d'une instruction. Le processeur est capable de détecter un grand nombre d'erreurs comme une division par zéro ou une opération sur une case mémoire ne disposant pas des bonnes permissions. Les exceptions sont classées en trois catégories : les *faults*, les *traps* et les *aborts*.

Chaque interruption et exception est identifiée par un nombre entier appelé vecteur d'interruption. Quand une interruption ou une exception est déclenchée, l'exécution passe du mode utilisateur au mode noyau et le système d'exploitation se réfère à la table des vecteurs d'interruptions (notée IDT pour *Interrupt Descriptor Table*). Ce tableau associe chaque vecteur d'interruption à l'adresse mémoire de la routine d'interruption (*Interrupt Service Routine*) à exécuter quand une interruption est levée. L'IDT est fourni par le système d'exploitation, par exemple dans le cas de Windows le vecteur d'interruption 0 est associé à l'adresse de la fonction *KiDivideErrorFault*.

Pour résumer, nous avons les interruptions et exceptions suivantes :

- les **interruptions matérielles** : elles sont générées par les périphériques extérieurs au processeur comme le clavier ;
- les **interruptions logicielles** : elles sont demandées par le logiciel en cours d'exécution et généralement utilisées pour les appels systèmes (dans le cas de l'architecture x86, elles sont déclenchées grâce à l'instruction `INT n`) ;
- les **exceptions** : elles sont déclenchées par le processeur lui-même lors d'une erreur. On retrouve dans cette catégorie les erreurs déclenchées par une division par zéro ou par une opération sur une

case mémoire dont les permissions ne sont pas suffisantes.

A.3.1 Le cas de Windows

Dans ce document, nous nous intéressons essentiellement au cas de Windows ainsi qu'aux exceptions et interruptions logicielles que nous appellerons simplement exceptions. De plus, nous nous plaçons exclusivement dans le cas où une exception se déclenche durant l'exécution d'un fichier binaire.

Lorsqu'une exception se produit, Windows démarre une procédure afin de pouvoir la traiter.

Tout d'abord, le processeur arrête l'exécution à l'endroit où l'exception s'est produite et transfère le contrôle au système d'exploitation. Ce dernier enregistre à la fois l'état machine du *thread* en cours et les informations qui décrivent l'exception. L'état machine du *thread* (c'est-à-dire la valeur des registres CPU au moment de l'exception) est sauvegardé dans une structure `_CONTEXT`. Ces informations (aussi appelées *context record*) permettent au système, dans le cas où l'exception est correctement traitée, de reprendre l'exécution du programme à l'endroit où l'exception a eu lieu. Pour cela, le système n'a qu'à charger dans chaque registre CPU la valeur correspondante présente dans le *context record*. Les informations relatives à l'exception en elle-même (appelée *exception record*) sont sauvegardées dans une structure `_EXCEPTION_RECORD`. Cette structure contient plusieurs informations comme le code qui identifie l'exception en cours ou l'adresse de l'instruction qui a déclenché l'exception.

Ensuite, le système tente de trouver un gestionnaire d'exception pour traiter l'exception. Il passe d'abord la main au gestionnaire d'exception le plus « proche » de là où l'exception a eu lieu. Si ce gestionnaire est capable de traiter l'exception alors il le fait, sinon la main est passée au gestionnaire de niveau inférieur et ainsi de suite jusqu'à trouver le gestionnaire qui va traiter l'exception en cours. Notons que si aucun gestionnaire du programme n'a pu traiter l'exception, Windows appelle la fonction `ExitProcess()` pour quitter l'application.

Lorsque le gestionnaire d'exception est exécuté, il dispose de l'*exception record* afin de pouvoir déterminer s'il a la capacité de gérer l'exception en cours. De plus, il a accès au *context record* qu'il peut modifier pour « réparer » l'erreur ayant déclenché l'exception.

Finalement, la main est rendue au système d'exploitation qui va pouvoir restaurer le *context record* et continuer l'exécution du programme.

A.3.2 Enregistrement d'un gestionnaire d'exception en assembleur

En pratique, les gestionnaires d'exceptions sont déclarés via une structure `_EXCEPTION_REGISTRATION`, donnée en listing A.1. Cette structure est stockée sur la pile sous la forme d'une liste chaînée.

Listing A.1 – Structure `_EXCEPTION_REGISTRATION`

```
_EXCEPTION_REGISTRATION struc
    prev    dd      ?
    handler dd      ?
_EXCEPTION_REGISTRATION ends
```

Le champ `prev` est un pointeur sur le précédent gestionnaire d'exception de la chaîne et `handler` pointe sur la fonction qui doit être appelée lorsqu'une exception est levée.

L'adresse de début de cette liste chaînée est stockée dans le premier champ de la structure appelée *Thread Environment Block* (TEB). Cette structure contient des informations concernant le *thread* courant. On accède à cette structure via le registre `FS` qui pointe sur la première adresse du TEB. L'adresse de début de la chaîne des gestionnaires d'exception étant le premier champ du TEB, elle est égale à `fs:[0x00]`. Il est à noter que `fs:[0x00]` pointe toujours sur le dernier gestionnaire d'exception installé qui est donc le premier à être appelé en cas d'exception.

Le code assembleur donné en listing A.2 permet de mettre en place un gestionnaire d'exception.

Listing A.2 – Enregistrement d'un gestionnaire d'exception

```
push handler          ; push handler address that we want to register
push fs:[0x00]        ; push previous handler record address
mov fs:[0x00], esp    ; set our new handler record as the first chain element
```

Bibliographie

[1] Hojjat Aghakhani, Fabio Gritti, Francesco Mecca, Martina Lindorfer, Stefano Ortolani, Davide Balzarotti, Giovanni Vigna et Christopher Kruegel. « When Malware Is Packin' Heat ; Limits of Machine Learning Classifiers Based on Static Analysis Features ». In : *Proceedings 2020 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. San Diego, CA : Internet Society, 2020. ISBN : 978-1-891562-61-7. DOI : 10.14722/ndss.2020.24310. URL : https://www.ndss-symposium.org/wp-content/uploads/2020/02/24310.pdf (visité le 06/11/2020).

[2] Andrew Appel. *Tiger Book - Modern Compiler Implementation in Java, 2nd Ed.* 2004.

[3] A. Arora, S. Garg et S. K. Peddoju. « Malware Detection Using Network Traffic Analysis in Android Based Mobile Devices ». In : *2014 Eighth International Conference on Next Generation Mobile Apps, Services and Technologies*. 2014 Eighth International Conference on Next Generation Mobile Apps, Services and Technologies. Sept. 2014, p. 66-71. DOI : 10.1109/NGMAST.2014.57.

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu et Irene Finocchi. « A Survey of Symbolic Execution Techniques ». In : *ACM Computing Surveys* 51.3 (2018).

[5] Daniel Balouek, Alexandra Carpen-Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lebre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr et Luc Sarzyniec. « Adding Virtualization Capabilities to Grid'5000 ». In : (), p. 24.

[6] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham et Alexander Pretschner. « Code Obfuscation Against Symbolic Execution Attacks ». In : *Proceedings of the 32Nd Annual Conference on Computer Security Applications*. ACSAC '16. New York, NY, USA : ACM, 2016, p. 189-200. ISBN : 978-1-4503-4771-6. DOI : 10.1145/2991079.2991114. URL : http://doi.acm.org/10.1145/2991079.2991114 (visité le 22/10/2018).

[7] Piotr Bania. *Generic Unpacking of Self-Modifying, Aggressive, Packed Binary Programs*. 28 mai 2009. arXiv : 0905.4581 [cs]. URL : http://arxiv.org/abs/0905.4581 (visité le 06/11/2020).

[8] Sebastien Bardin, Robin David et Jean-Yves Marion. « Backward-Bounded DSE : Targeting Infeasibility Questions on Obfuscated Codes ». In : *2017 IEEE Symposium on Security and Privacy (SP)*. 2017 IEEE Symposium on Security and Privacy (SP). San Jose, CA, USA : IEEE, mai 2017, p. 633-651. ISBN : 978-1-5090-5533-3. DOI : 10.1109/SP.2017.36. URL : http://ieeexplore.ieee.org/document/7958602/ (visité le 21/10/2020).

[9] Sebastien Bardin, Philippe Herrman, Jérôme Leroux, Olivier Ly, Renaud Tabary et Aymeric Vincent. « The BINCOA Framework for Binary Code Analysis ». In : *Computer Aided Verification*. Sous la dir. de Gopalakrishnan, Ganesh, Qadeer et Shaz. T. 6806. United Kingdom, 2011, p. 165-170. DOI : 10.1007/978-3-642-22110-1_13. URL : https://hal.archives-ouvertes.fr/hal-01006499 (visité le 27/10/2020).

[10] Sébastien Bardin, Philippe Herrmann et Franck Védrine. « Refinement-Based CFG Reconstruction from Unstructured Programs ». In : *Verification, Model Checking, and Abstract Interpretation*. Sous la dir. de Ranjit Jhala et David Schmidt. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, p. 54-69. ISBN : 978-3-642-18275-4.

[11] Clark Barrett, Pascal Fontaine et Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. 2016.

[12] Erick Bauman, Zhiqiang Lin et Kevin W. Hamlen. « Superset Disassembly : Statically Rewriting X86 Binaries Without Heuristics ». In : *Proceedings 2018 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. San Diego, CA : Internet Society, 2018. ISBN : 978-1-891562-49-5. DOI : 10.14722/ndss.2018.23300. URL : https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_05A-4_Bauman_paper.pdf (visité le 21/10/2020).

[13] Z. Bazrafshan, H. Hashemi, S. M. H. Fard et A. Hamzeh. « A Survey on Heuristic Malware Detection Techniques ». In : *The 5th Conference on Information and Knowledge Technology*. The 5th Conference on Information and Knowledge Technology. Mai 2013, p. 113-120. DOI : 10.1109/IKT.2013.6620049.

[14] Fabrice Bellard. « QEMU, a Fast and Portable Dynamic Translator ». In : *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA : USENIX Association, 10 avr. 2005, p. 41.

[15] Guillaume Bonfante, Jose Fernandez, Jean-Yves Marion, Benjamin Rouxel, Fabrice Sabatier et Aurélien Thierry. « CoDisasm : Medium Scale Concatic Disassembly of Self-Modifying Binaries with Overlapping Instructions ». In : 22nd ACM Conference on Computer and Communications Security. 12 oct. 2015. DOI : 10.1145/2810103.2813627. URL : https://hal.inria.fr/hal-01257908 (visité le 05/11/2020).

[16] *Brain - The First Computer Virus Was Created by Two Brothers from Pakistan. They Just Wanted to Prevent Their Customers of Making Illegal Software Copies*. The Vintage News. 8 sept. 2016. URL : /2016/09/08/priority-brain-first-computer-virus-created-two-brothers-pakistan-just-wanted-prevent-customers-making-illegal-software-copies/ (visité le 17/11/2020).

[17] David Brumley, Ivan Jager, Thanassis Avgerinos et Edward J. Schwartz. « BAP : A Binary Analysis Platform ». In : *Computer Aided Verification*. Sous la dir. de Ganesh Gopalakrishnan et Shaz Qadeer. Lecture Notes in Computer Science. Berlin, Heidelberg : Springer, 2011, p. 463-469. ISBN : 978-3-642-22110-1. DOI : 10.1007/978-3-642-22110-1_37.

[18] Cristian Cadar, Daniel Dunbar et Dawson Engler. « KLEE : Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs ». In : *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California : USENIX Association, 8 déc. 2008, p. 209-224.

[19] Joan Calvet, Fanny Lalonde Lévesque, José M Fernandez, Erwann Traourouder, François Menet et Jean-Yves Marion. « WaveAtlas : Surfing Through the Landscape of Current Malware Packers ». In : (2015), p. 7.

[20] Binlin Cheng, Jiang Ming, Jianmin Fu, Guojun Peng, Ting Chen, Xiaosong Zhang et Jean-Yves Marion. « Towards Paving the Way for Large-Scale Windows Malware Analysis : Generic Binary Unpacking with Orders-of-Magnitude Performance Boost ». In : *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Toronto, Canada : Association for Computing Machinery, 15 jan. 2018, p. 395-411. ISBN : 978-1-4503-5693-0. DOI : 10.1145/3243734.3243771. URL : https://doi.org/10.1145/3243734.3243771 (visité le 13/05/2020).

[21] Christian Collberg, Clark Thomborson et Douglas Low. « A Taxonomy of Obfuscating Transformations ». In : *http ://www.cs.auckland.ac.nz/staff-cgi-bin/mjd/csTRcgi.pl ?serial* (1er jan. 1997).

[22] Christian Collberg, Clark Thomborson et Douglas Low. « Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs ». In : *Conference Record of the Annual ACM Symposium on Principles of Programming Languages* 184–196 (21 nov. 1997). DOI : 10.1145/268946.268962.

[23] Daniele Cono D'Elia, Emilio Coppa, Simone Nicchi, Federico Palmaro et Lorenzo Cavallaro. « SoK : Using Dynamic Binary Instrumentation for Security (And How You May Get Caught Red Handed) ». In : *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. Asia CCS '19 : ACM Asia Conference on Computer and Communications Security. Auckland New Zealand : ACM, 2 juil. 2019, p. 15-27. ISBN : 978-1-4503-6752-3. DOI : 10.1145/3321705.3329819. URL : https://dl.acm.org/doi/10.1145/3321705.3329819 (visité le 18/11/2020).

[24] Mila Dalla Preda, Roberto Giacobazzi, Saumya Debray, Kevin Coogan et Gregg M. Townsend. « Modelling Metamorphism by Abstract Interpretation ». In : *Static Analysis*. Sous la dir. de Radhia Cousot et Matthieu Martel. Lecture Notes in Computer Science. Berlin, Heidelberg : Springer, 2010, p. 218-235. ISBN : 978-3-642-15769-1. DOI : 10.1007/978-3-642-15769-1_14.

[25] Robin David. « Formal Approaches for Automatic Deobfuscation and Reverse-Engineering of Protected Codes ». Université de Lorraine, 6 jan. 2017.

[26] *Dependency Walker*. URL : https://www.dependencywalker.com/ (visité le 14/11/2020).

[27] Sushant Dinesh, Nathan Burow, Dongyan Xu et Mathias Payer. « RetroWrite : Statically Instrumenting COTS Binaries for Fuzzing and Sanitization ». In : *2020 IEEE Symposium on Security and Privacy (SP)*. 2020 IEEE Symposium on Security and Privacy (SP). San Francisco, CA, USA : IEEE, mai 2020, p. 1497-1511. ISBN : 978-1-72813-497-0. DOI : 10.1109/SP40000.2020.00009. URL : https://ieeexplore.ieee.org/document/9152762/ (visité le 21/10/2020).

[28] Adel Djoudi et Sébastien Bardin. « BINSEC : Binary Code Analysis with Low-Level Regions ». In : (2015), p. 6.

[29] Adel Djoudi et Sebastien Bardin. « An Extended Version of DBA ». In : (), p. 21.

[30] Gina-Adriana Dobrescu et Maxime Meignan. « Binary Analysis and Dynamic Jumps ». In : 2014.

[31] Thomas Dullien et Sebastian Porst. « REIL : A Platform-Independent Intermediate Representation of Disassembled Code for Static Code Analysis ». In : (), p. 7.

[32] Mattias Felleisen. « The Theory and Practice of First-Class Prompts ». In : *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '88. San Diego, California, USA : Association for Computing Machinery, 13 jan. 1988, p. 180-190. ISBN : 978-0-89791-252-5. DOI : 10.1145/73560.73576. URL : https://doi.org/10.1145/73560.73576 (visité le 08/12/2020).

[33] Daniel Gibert, Carles Mateu et Jordi Planes. « The Rise of Machine Learning for Detection and Classification of Malware : Research Developments, Trends and Challenges ». In : *Journal of Network and Computer Applications* 153 (1er mar. 2020), p. 102526. ISSN : 1084-8045. DOI : 10.1016/j.jnca.2019.102526. URL : http://www.sciencedirect.com/science/article/pii/S1084804519303868 (visité le 18/11/2020).

[34] Patrice Godefroid, Nils Klarlund et Koushik Sen. « DART : Directed Automated Random Testing ». In : *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05. Chicago, IL, USA : Association for Computing Machinery, 12 juin 2005, p. 213-223. ISBN : 978-1-59593-056-9. DOI : 10.1145/1065010.1065036. URL : https://doi.org/10.1145/1065010.1065036 (visité le 08/12/2020).

[35] Fanglu Guo, Peter Ferrie et Tzi-cker Chiueh. « A Study of the Packer Problem and Its Solutions ». In : *Recent Advances in Intrusion Detection*. Sous la dir. de Richard Lippmann, Engin Kirda et Ari Trachtenberg. T. 5230. Lecture Notes in Computer Science. Berlin, Heidelberg : Springer Berlin Heidelberg, 2008, p. 98-115. ISBN : 978-3-540-87402-7 978-3-540-87403-4. DOI : 10.1007/978-3-540-87403-4_6. URL : http://link.springer.com/10.1007/978-3-540-87403-4_6 (visité le 28/10/2020).

[36] *Hex-Rays IDA Disassembler*. URL : https://www.hex-rays.com/products/ida/ (visité le 21/11/2018).

[37] Xin Hu, Kang G Shin, Sandeep Bhatkar et Kent Griffin. « MutantX-S : Scalable Malware Clustering Based on Static Features ». In : (2013), p. 13.

[38] « Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B, 2C & 2D) : Instruction Set Reference, A-Z ». In : (), p. 2198.

[39] Bartosz Jasiul, Joanna Śliwa, Kamil Gleba et Marcin Szpyrka. « Identification of Malware Activities with Rules ». In : 2014 Federated Conference on Computer Science and Information Systems. 29 sept. 2014, p. 101-110. doi : 10.15439/2014F265. url : https://fedcsis.org/proceedings/2014/drp/265.html (visité le 18/11/2020).

[40] Neil Jones. *Computability and Complexity From a Programming Perspective*.

[41] Min Gyung Kang, Pongsin Poosankam et Heng Yin. « Renovo : A Hidden Code Extractor for Packed Executables ». In : *Proceedings of the 2007 ACM Workshop on Recurring Malcode - WORM '07*. The 2007 ACM Workshop. Alexandria, Virginia, USA : ACM Press, 2007, p. 46. isbn : 978-1-59593-886-2. doi : 10.1145/1314389.1314399. url : http://portal.acm.org/citation.cfm?doid=1314389.1314399 (visité le 23/10/2020).

[42] Johannes Kinder et Helmut Veith. « Jakstab : A Static Analysis Platform for Binaries ». In : *Proceedings of the 20th International Conference on Computer Aided Verification*. CAV '08. Berlin, Heidelberg : Springer-Verlag, 2008, p. 423-427. isbn : 978-3-540-70543-7. doi : 10.1007/978-3-540-70545-1_40. url : http://dx.doi.org/10.1007/978-3-540-70545-1_40 (visité le 26/03/2018).

[43] Johannes Kinder, Florian Zuleger et Helmut Veith. « An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries ». In : *Verification, Model Checking, and Abstract Interpretation*. International Workshop on Verification, Model Checking, and Abstract Interpretation. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 18 jan. 2009, p. 214-228. isbn : 978-3-540-93899-6 978-3-540-93900-9. doi : 10.1007/978-3-540-93900-9_19. url : https://link.springer.com/chapter/10.1007/978-3-540-93900-9_19 (visité le 20/03/2018).

[44] James C. King. « Symbolic Execution and Program Testing ». In : *Communications of the ACM* 19.7 (1er juil. 1976), p. 385-394. issn : 0001-0782. doi : 10.1145/360248.360252. url : https://doi.org/10.1145/360248.360252 (visité le 07/12/2020).

[45] David Korczynski. « RePEconstruct : Reconstructing Binaries with Self-Modifying Code and Import Address Table Destruction ». In : *2016 11th International Conference on Malicious and Unwanted Software (MALWARE)*. 2016 11th International Conference on Malicious and Unwanted Software (MALWARE). Oct. 2016, p. 1-8. doi : 10.1109/MALWARE.2016.7888727.

[46] « Le piratage de TV5 Monde vu de l'intérieur ». In : *Le Monde.fr* (10 juin 2017). url : https://www.lemonde.fr/pixels/article/2017/06/10/le-piratage-de-tv5-monde-vu-de-l-interieur_5142046_4408996.html (visité le 17/11/2020).

[47] Xavier Leroy et Sandrine Blazy. « Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations ». In : *Journal of Automated Reasoning* 41.1 (2008), p. 1-31. doi : 10.1007/s10817-008-9099-0. url : https://hal.inria.fr/inria-00289542 (visité le 28/09/2020).

[48] *Loveletter*. url : http://virus.wikidot.com/loveletter (visité le 17/11/2020).

[49] Sebastiano Mariani, Lorenzo Fontana, Fabio Gritti et Stefano D'Alessio. « PinDemonium : A DBI-Based Generic Unpacker for Windows Executables ». In : *Black Hat USA* (2016), p. 59.

[50] Lorenzo Martignoni, Mihai Christodorescu et Somesh Jha. « OmniUnpack : Fast, Generic, and Safe Unpacking of Malware ». In : (2007), p. 10.

[51] Xiaozhu Meng et Barton P. Miller. « Binary Code Is Not Easy ». In : *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ISSTA 2016. New York, NY, USA : ACM, 2016, p. 24-35. isbn : 978-1-4503-4390-9. doi : 10.1145/2931037.2931047. url : http://doi.acm.org/10.1145/2931037.2931047 (visité le 22/10/2018).

[52] Lakshman Nataraj. *Nearly 70% of Packed Windows System Files Are Labeled as Malware*. UCSB Sarvam Blog. 2013. URL : https://sarvamblog.blogspot.com/2013/05/nearly-70-of-packed-windows-system.html (visité le 06/11/2020).

[53] Minh Hai Nguyen, Thien Binh Nguyen, Thanh Tho Quan et Mizuhito Ogawa. « A Hybrid Approach for Control Flow Graph Construction from Binary Code ». In : *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*. 2013 20th Asia-Pacific Software Engineering Conference (APSEC). T. 2. Déc. 2013, p. 159-164. DOI : 10.1109/APSEC.2013.132.

[54] Matt Pietrek. *A Crash Course on the Depths of Win32 Structured Exception Handling*. 1997. URL : /paper/A-crash-course-on-the-depths-of-Win32-structured-Pietrek/800f113c0c78e52010097741fd49d (visité le 06/11/2020).

[55] Mila Dalla Preda, Roberto Giacobazzi, Matias Madou et Koen De Bosschere. « Opaque Predicates Detection by Abstract Interpretation ». In : *In Proc. Internat. Conf on Algebraic Methodology and Software Technology (AMAST'06*. Springer-Verlag, 2006, p. 35-50.

[56] *Programme informatique*. In : *Wikipédia*. 11 nov. 2020. URL : https://fr.wikipedia.org/w/index.php?title=Programme_informatique&oldid=176496718 (visité le 24/11/2020).

[57] Jing Qiu, Xiao Hong Su et Pei Jun Ma. « Deobfuscate Non-Returning Calls and Call-Stack Tampering in Instruction Traces ». In : *Advanced Materials Research* 989-994 (2014), p. 1782-1785. ISSN : 1662-8985. DOI : 10.4028/www.scientific.net/AMR.989-994.1782. URL : https://www.scientific.net/AMR.989-994.1782 (visité le 06/03/2018).

[58] *Recrudescence d'activité Emotet En France – CERT-FR*. URL : https://www.cert.ssi.gouv.fr/alerte/CERTFR-2020-ALE-019/ (visité le 19/11/2020).

[59] Daniel Reynaud. « Analyse de Codes Auto-Modifiants Pour La Sécurité Logicielle ». These de doctorat. Vandoeuvre-les-Nancy, INPL, 15 oct. 2010. URL : https://www.theses.fr/2010INPL049N (visité le 11/12/2020).

[60] Kevin A. Roundy et Barton P. Miller. « Hybrid Analysis and Control of Malware ». In : *Recent Advances in Intrusion Detection*. Sous la dir. de Somesh Jha, Robin Sommer et Christian Kreibich. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, p. 317-338. ISBN : 978-3-642-15512-3.

[61] P. Royal, M. Halpin, D. Dagon, R. Edmonds et W. Lee. « PolyUnpack : Automating the Hidden-Code Extraction of Unpack-Executing Malware ». In : *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*. 2006 22nd Annual Computer Security Applications Conference (ACSAC'06). Déc. 2006, p. 289-300. DOI : 10.1109/ACSAC.2006.38.

[62] Amr Sabry et Matthias Felleisen. « Reasoning about Programs in Continuation-Passing Style. » In : *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*. LFP '92. San Francisco, California, USA : Association for Computing Machinery, 1er jan. 1992, p. 288-298. ISBN : 978-0-89791-481-9. DOI : 10.1145/141471.141563. URL : https://doi.org/10.1145/141471.141563 (visité le 08/12/2020).

[63] Jonathan Salwan. « L'usage de l'exécution symbolique pour la déobfuscation binaire en milieu industriel ». Grenoble : Unviersité Grenoble Alpes, 13 fév. 2020.

[64] B. Schwarz, S. Debray et G. Andrews. « Disassembly of Executable Code Revisited ». In : *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.* Ninth Working Conference on Reverse Engineering, 2002. Proceedings. 2002, p. 45-54. DOI : 10.1109/WCRE.2002.1173063.

[65] Koushik Sen, Darko Marinov et Gul Agha. « CUTE : A Concolic Unit Testing Engine for C ». In : *ACM SIGSOFT Software Engineering Notes* 30.5 (1er sept. 2005), p. 263-272. ISSN : 0163-5948. DOI : 10.1145/1095430.1081750. URL : https://doi.org/10.1145/1095430.1081750 (visité le 09/12/2020).

[66] Monirul Sharif, Andrea Lanzi, Jonathon Giffin et Wenke Lee. « Impeding Malware Analysis Using Conditional Code Obfuscation ». In : (), p. 13.

[67] Monirul Sharif, Vinod Yegneswaran, Hassen Saidi, Phillip Porras et Wenke Lee. « Eureka : A Framework for Enabling Static Malware Analysis ». In : *Computer Security - ESORICS 2008*. Sous la dir. de Sushil Jajodia et Javier Lopez. T. 5283. Lecture Notes in Computer Science. Berlin, Heidelberg : Springer Berlin Heidelberg, 2008, p. 481-500. ISBN : 978-3-540-88312-8 978-3-540-88313-5. DOI : 10.1007/978-3-540-88313-5_31. URL : http://link.springer.com/10.1007/978-3-540-88313-5_31 (visité le 28/10/2020).

[68] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel et Giovanni Vigna. « (State of) The Art of War : Offensive Techniques in Binary Analysis ». In : (), p. 20.

[69] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam et Prateek Saxena. « BitBlaze : A New Approach to Computer Security via Binary Analysis ». In : *Information Systems Security*. Sous la dir. de R. Sekar et Arun K. Pujari. T. 5352. Lecture Notes in Computer Science. Berlin, Heidelberg : Springer Berlin Heidelberg, 2008, p. 1-25. ISBN : 978-3-540-89861-0 978-3-540-89862-7. DOI : 10.1007/978-3-540-89862-7_1. URL : http://link.springer.com/10.1007/978-3-540-89862-7_1 (visité le 26/10/2020).

[70] Daniele Ucci, Leonardo Aniello et Roberto Baldoni. « Survey of Machine Learning Techniques for Malware Analysis ». In : *Computers & Security* 81 (1er mar. 2019), p. 123-147. ISSN : 0167-4048. DOI : 10.1016/j.cose.2018.11.001. URL : http://www.sciencedirect.com/science/article/pii/S0167404818303808 (visité le 18/11/2020).

[71] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos et Pablo G. Bringas. « SoK : Deep Packer Inspection : A Longitudinal Study of the Complexity of Run-Time Packers ». In : *2015 IEEE Symposium on Security and Privacy*. 2015 IEEE Symposium on Security and Privacy (SP). San Jose, CA : IEEE, mai 2015, p. 659-673. ISBN : 978-1-4673-6949-7. DOI : 10.1109/SP.2015.46. URL : https://ieeexplore.ieee.org/document/7163053/ (visité le 26/10/2020).

[72] *VirusTotal*. URL : https://www.virustotal.com/ (visité le 19/11/2020).

[73] Fish Wang et Yan Shoshitaishvili. « Angr - The Next Generation of Binary Analysis ». In : *2017 IEEE Cybersecurity Development (SecDev)*. 2017 IEEE Cybersecurity Development (SecDev). Sept. 2017, p. 8-9. DOI : 10.1109/SecDev.2017.14.

[74] Xinyu Wang, Jun Sun, Zhenbang Chen, Peixin Zhang, Jingyi Wang et Yun Lin. « Towards Optimal Concolic Testing ». In : *Proceedings of the 40th International Conference on Software Engineering*. ICSE '18. Gothenburg, Sweden : Association for Computing Machinery, 27 mai 2018, p. 291-302. ISBN : 978-1-4503-5638-1. DOI : 10.1145/3180155.3180177. URL : https://doi.org/10.1145/3180155.3180177 (visité le 07/12/2020).

[75] *WannaCry*. In : *Wikipédia*. 4 nov. 2020. URL : https://fr.wikipedia.org/w/index.php?title=WannaCry&oldid=176244728 (visité le 17/11/2020).

[76] W. Yan, Z. Zhang et N. Ansari. « Revealing Packed Malware ». In : *IEEE Security Privacy* 6.5 (sept. 2008), p. 65-69. ISSN : 1558-4046. DOI : 10.1109/MSP.2008.126.

[77] Yongxin Zhou, Alec Main, Yuan X. Gu et Harold Johnson. « Information Hiding in Software with Mixed Boolean-Arithmetic Transforms ». In : *Information Security Applications*. Sous la dir. de Sehun Kim, Moti Yung et Hyung-Woo Lee. Lecture Notes in Computer Science. Berlin, Heidelberg : Springer, 2007, p. 61-75. ISBN : 978-3-540-77535-5. DOI : 10.1007/978-3-540-77535-5_5.

Résumé

L'augmentation des cyberattaques dans le monde fait de l'analyse des codes malveillants un domaine de recherche prioritaire. Ces logiciels utilisent diverses méthodes de protection, encore appelées obfuscations, visant à contourner les antivirus et à ralentir le travail d'analyse. Dans ce contexte, cette thèse apporte une solution à la construction du Graphe de Flot de Contrôle (GFC) d'un code binaire obfusqué. Nous avons développé la plateforme BOA (*Basic blOck Analysis*) qui effectue une analyse statique d'un code binaire protégé. Pour cela, nous avons défini une sémantique s'appuyant sur l'outil BINSEC à laquelle nous avons ajouté des continuations. Ces dernières permettent d'une part de contrôler les auto-modifications, et d'autre part de simuler le système d'exploitation pour traiter les appels et interruptions système. L'analyse statique est faite en exécutant symboliquement le code binaire et en calculant les valeurs des états du système à l'aide de solveur SMT. Ainsi, nous effectuons une analyse du flot de données afin de construire le GFC en calculant les adresses de transfert. Enfin, la gestion des boucles est réalisée en transformant un GFC en un automate à pile. BOA est capable de calculer les adresses des sauts dynamiques, de détecter les prédicats opaques, de calculer les adresses de retour sur une pile même si elles ont été falsifiées, de gérer les falsifications des gestionnaires d'interruption, reconstruire à la volée les tables d'importation, et pour finir, de gérer les auto-modifications. Nous avons validé la correction de BOA en utilisant l'obfuscateur de code Tigress. Ensuite, nous avons testé BOA sur 35 packers connus et nous avons montré que dans 30 cas, BOA était capable de reconstruire complètement ou partiellement le binaire initialement masqué. Pour finir, nous avons détecté les prédicats opaques protégeant XTunnel, un *malware* utilisé lors des élections américaines de 2016, et nous avons partiellement dépacké un échantillon du cheval de Troie Emotet, qui, le 14/10/2020 n'était détecté que par 7 antivirus sur les 63 que propose VirusTotal. Ce travail contribue au développement des outils d'analyse statique des codes malveillants. Contrairement aux analyses dynamiques, cette solution permet une analyse sans exécution du binaire, ce qui offre un double avantage : d'une part une approche statique est plus facile à déployer, et d'autre part le code malveillant n'étant pas exécuté, il ne peut pas prévenir son auteur.

Mots-clés : Logiciel malveillant, Obfuscation, Flot de données, Exécution symbolique, Graphe de flot de contrôle.

Abstract

The increase in cyber attacks around the world makes malicious code analysis a priority research area. This software uses various protection methods, also known as obfuscations, to bypass antivirus software and slow down the analysis process. In this context, this thesis provides a solution to build the Control Float Graph (CFG) of obfuscated binary code. We developed the BOA platform (*Basic blOck Analysis*) which performs a static analysis of a protected binary code. For this, we have defined a semantics based on the BINSEC tool to which we have added continuations. These allow on one hand to control the self-modifications, and on the other hand to simulate the operating system to handle system calls and interruptions. The static analysis is done by symbolically executing the binary code and calculating the values of the system states using SMT solvers. Thus, we perform a data flow analysis to build the CFG by calculating the transfer addresses. Finally, loop handling is performed by transforming a CFG into a pushdown automaton. BOA is able to compute dynamic jump addresses, to detect opaque predicates, to compute return addresses on a stack even if they have been falsified, to manage interrupt handler falsifications, to rebuild import tables on the fly, and finally, to manage self-modifications. We validated the BOA correction using the Tigress code obfuscator. Then, we tested BOA on 35 known packers and showed that in 30 cases, BOA was able to completely or partially rebuild the initially protected binary. Finally, we detected the opaque predicates protecting XTunnel, a malware used during the 2016 U.S. elections, and we partially unpacked a sample of the Emotet Trojan, which on 14/10/2020 was detected by only 7 antivirus programs out of the 63 offered by VirusTotal. This work contributes to the development of tools for static analysis of malicious code. In contrast to dynamic methods, this solution allows an analysis without executing the binary, which offers a double advantage : on the one hand, a static approach is easier to deploy, and on the other hand, since the malicious code is not executed, it cannot warn its author.

Keywords: Malware, Obfuscation, Data flow, Symbolic execution, Control flow graph.