HAL
open science

Analyse de programmes malveillants par abstraction de comportements

Philippe Beaucamps

▶ **To cite this version:**

Philippe Beaucamps. Analyse de programmes malveillants par abstraction de comportements. Autre [cs.OH]. Institut National Polytechnique de Lorraine, 2011. Français. NNT : 2011INPL092N . tel-01751072

HAL Id: tel-01751072

https://hal.univ-lorraine.fr/tel-01751072

Submitted on 29 Mar 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE LORRAINE

AVERTISSEMENT

Ce document est le fruit d'un long travail approuvé par le jury de soutenance et mis à disposition de l'ensemble de la communauté universitaire élargie.

Il est soumis à la propriété intellectuelle de l'auteur. Ceci implique une obligation de citation et de référencement lors de l'utilisation de ce document.

D'autre part, toute contrefaçon, plagiat, reproduction illicite encourt une poursuite pénale.

Contact : ddoc-theses-contact@univ-lorraine.fr

LIENS

Code de la Propriété Intellectuelle. articles L 122. 4
Code de la Propriété Intellectuelle. articles L 335.2- L 335.10
http://www.cfcopies.com/V2/leg/leg_droi.php
http://www.culture.gouv.fr/culture/infos-pratiques/droits/protection.htm

Nancy-Université
INPL

Département de formation doctorale en informatique

École doctorale IAEM Lorraine

Analyse de Programmes Malveillants par Abstraction de Comportements

THÈSE

présentée et soutenue publiquement le 14 novembre 2011

pour l'obtention du

Doctorat de l'Institut National Polytechnique de Lorraine

(spécialité informatique)

par

Philippe Beaucamps

Rapporteurs :	Sandrine Blazy	Professeur, IRISA
	Sophie Tison	Professeur, INRIA & LIFL
Examinateurs :	Roberto Giacobazzi	Professeur, Università degli Studi di Verona, Italie
	Hélène Kirchner	Professeur, INRIA Bordeaux
	Jean-Yves Marion	Professeur, École des Mines de Nancy
	Isabelle Gnaedig	Chargée de recherche, INRIA Nancy

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

Loria
Laboratoire lorrain de recherche en informatique et ses applications

Table des matières

Table des matières **1**

1 Introduction **3**
1.1 Objectifs 5
1.2 Travaux existants 7
1.3 Contributions 11

2 Fondements **13**
2.1 Qu'est-ce qu'un Comportement ? 13
2.2 Définitions 15
2.3 Ensemble de Traces d'un Programme 17
2.4 Formalisation de la Détection Comportementale Classique . . . 23

3 Automates de Traces **29**
3.1 Construction par Analyse Dynamique 29
3.2 Construction par Analyse Statique 32
3.3 Exemple d'Automate de Traces (Java) : Fuite de SMS 46
3.4 Exemple d'Automate de Traces (C) : Keylogger 50

4 Abstraction par Réécriture de Mots **59**
4.1 Définitions 60
4.2 Behavior Patterns 61
4.3 Abstraction de Traces 63
4.4 Abstraction d'Ensembles Réguliers de Traces 67
4.5 Détection de Comportements 71
4.6 Expérimentations 73
4.7 Conclusion 79

5 Abstraction par Réécriture de Termes **81**
5.1 Définitions 83

5.2 Behavior Patterns 88
5.3 Problème de la Détection 89
5.4 Complexité de la Détection 99
5.5 Abstraction de Traces 102
5.6 Application à la Détection d'une Fuite d'Information 111
5.7 Expérimentations 113
5.8 Conclusion 120

6 Abstraction Pondérée **121**
6.1 Définitions 122
6.2 Abstraction Pondérée 127
6.3 Problème de la détection 133
6.4 Abstraction Rationnelle 145
6.5 Conclusion 156

7 Conclusion **157**
7.1 Applications 160
7.2 Perspectives 162

Bibliographie **165**

Chapitre 1

Introduction

Bouleversement du siècle dernier, les systèmes d'information et les réseaux ont conduit à une nouvelle typologie des risques et des menaces, requalifiés pour l'occasion avec le préfixe *cyber*, qui véhicule la notion d'univers de l'information numérique. De façon générale, pourtant, ils ont peu changé. Leur objectifs sont toujours le vol d'informations personnelles, la destruction de données, la perturbation de services, ... Leurs motivations sont toujours stratégiques, idéologiques, politiques, terroristes, cupides, ... [97] Leurs cibles sont toujours les particuliers, les entreprises, les organismes gouvernementaux, ... Pourtant, leur volume et leur existence virtuelle ont rendu toutes les mesures habituelles de protection inadaptées. Et loin d'être progressivement maîtrisés, ils semblent plutôt échapper à tout contrôle.

Ainsi, le nombre de codes malveillants découverts quotidiennement est en hausse constante [84, 19], de nouvelles attaques d'entreprises défraient régulièrement la chronique (Sony, RSA, Opération Aurora, ...) et des attaques d'organismes gouvernementaux (comme les Ministères des Finances canadien et français) et de pays (dénis de service contre l'Estonie en 2007 puis contre la Géorgie en 2008) sont également de plus en plus fréquents. Ces risques croissants sont entretenus par le développement d'Internet (2 milliards de personnes connectées fin 2010 selon l'ITU, Union Internationale des Télécommunications), par l'interconnexion croissante de tous les appareils et de tous les services, et par le développement des outils de communication mobiles, qui ne tirent que partiellement les leçons du passé. Enfin, la professionnalisation de ces attaques, qu'elles soient orchestrées par des réseaux mafieux ou par des gouvernements, et leur démocratisation, illustrée par l'opération "Payback" invitant les sympathisants du site Wikileaks à participer aux représailles contre les entreprises Mastercard, Visa, Paypal et Amazon, aggrave encore la situation.

Aussi, la question de la sécurité des systèmes d'information est-elle prégnante et pleine de défis : sa nature virtuelle rend obsolètes des mesures de protection pourtant éprouvées, sa nature distribuée rend caduques les stratégies habituelles de contrôle, sa nature anonyme rend ineffectives les stratégies habituelles de représailles et de dissuasion, sa nature imprévisible demande une vigilance constante, et enfin n'importe qui, jusqu'aux personnes les plus haut placées, peut devenir acteur inconscient de sa propre compromission.

Dans le même temps, les développements technologiques récents créent de nouveaux vecteurs d'attaques. En particulier, la prépondérance de plus en plus marquée de l'informatique Web et mobile, et son accès de plus en plus vaste à des données sensibles accentuent le problème de la sécurisation de ces données.

Les attaques via un navigateur sont courantes et variées. Elles exploitent des failles du navigateur pour obtenir l'accès à la machine de l'utilisateur. Elles simulent la page d'un site bancaire pour obtenir des données sensibles (phishing et tabnabbing). Elles exploitent des erreurs de programmation d'un site en injectant un script dans une page Web, afin de se faire passer pour l'utilisateur et ainsi accéder à l'ensemble des ses données (XSS). Elles forgent des requêtes vers des sites Web afin par exemple de désactiver une protection ou de faire une demande de virement et elles cachent l'URL ainsi construit dans une image, de telle sorte que le navigateur exécute automatiquement la requête au nom du client (CSRF).

Quant aux attaques contre les appareils mobiles, elles profitent de l'engouement récent pour ces terminaux, dans les pays développés comme ceux en voie de développement, et de pratiques peu sécurisées. Sans le poids de devoir composer avec la présence de solutions de sécurité, les opportunités de cybercriminalité sont vastes. Sur certains points, elles sont similaires à celles des PCs, avec par exemple la mise en place de botnets mobiles, l'envoi massif de spam, la collecte d'informations personnelles (mots de passe, codes bancaires), la collecte de carnets de contacts, mais elles ont aussi quelques spécificités, avec notamment la monétisation par envoi de SMS surtaxés. Ainsi, 50 applications mobiles malicieuses ont été découvertes récemment sur la place de marché officielle du système Android. Elles avaient été installées par plus de 50.000 utilisateurs, collectaient des données personnelles et installaient une backdoor dans le système.

Pourtant, aucune solution n'est encore parvenue à s'imposer pour contrer une attaque informatique. Et pour cause, les obstacles sont multiples, techniques comme humains, et toute solution reste finalement limitée par la faillibilité de son utilisateur et par sa propre faillibilité. Deux approches prévalent néanmoins. Soit la sécurité est gérée par le système lui-même, soit elle est gérée

par une solution extérieure au système, chargée de contrôler et de restreindre l'usage du système. Dans cette seconde approche, certaines solutions procèdent par *analyse du code* d'un programme, en identifiant les menaces grâce à une connaissance a priori de leur composition et de leurs caractéristiques tandis que d'autres procèdent par *analyse du comportement* d'un programme, en tentant d'identifier une menace par l'analyse et le croisement du comportement du programme avec un référentiel prédéfini de comportements sains et malsains. Ainsi, les solutions reposant sur une analyse du code d'un programme permettent d'identifier efficacement des menaces connues ou héritant directement de menaces connues, la détection étant alors réactive. Et les solutions par analyse du comportement d'un programme permettent de compléter les premières solutions lorsqu'il s'agit de découvrir de nouvelles menaces, susceptibles de se trahir par leur comportement, la détection étant alors proactive.

1.1 Objectifs

Dans cette thèse, nous nous intéressons plus particulièrement à l'analyse comportementale de programme. Celle-ci peut être abordée de deux manières :

Analyse dynamique Une trace d'exécution spécifique d'un programme est analysée. On rencontre cette situation dans le cas d'une surveillance du système en temps réel ou lorsque le code est émulé ou bien exécuté dans un environnement confiné. Néanmoins, cette approche a deux inconvénients. D'une part, une seule trace n'est pas représentative du code et une analyse négative ne garantit en rien que le code soit sain. D'autre part, en n'analysant que le comportement passé, cette approche est souvent insuffisante pour endiguer à temps une activité malicieuse.

Analyse statique Le comportement d'un programme est analysé dans son intégralité. Dans ce cas, l'analyse est plus exhaustive et par extension plus sûre que dans le cas dynamique. Par ailleurs, elle permet de prendre des décisions en amont d'un comportement critique. Néanmoins, elle comporte deux difficultés : une représentation intelligible du code permettant de déterminer le comportement du programme n'est pas toujours disponible, en particulier lorsque le code est binaire et protégé (packing, obfuscation, etc.), et l'analyse peut être plus coûteuse puisqu'elle porte sur l'ensemble des exécutions possibles du code.

Ainsi, l'analyse statique est plus puissante que son pendant dynamique. En outre, en dépit de ses limitations, son application est facilitée dans un certain nombre de scénarios. Tout d'abord, dans le cas d'un programme se protégeant lui-même, l'analyse peut n'être lancée qu'au moment d'une activité critique,

moment auquel le programme malveillant a en général désactivé toutes ses protections.

Ensuite, l'évolution récente des plateformes de développement, qui favorise une adoption croissante de langages de haut-niveau, notamment pour les applications tierce partie pour les mobiles [1] et pour le Web, est particulièrement propice à l'analyse statique. Au vu du nombre croissant d'applications malicieuses sur ces systèmes, l'analyse statique, en particulier comportementale, paraît donc trouver un cadre d'application naturel afin de valider une application préalablement à sa publication sur la place de marché.

Par ailleurs, le modèle de sécurité dans lequel ces applications s'exécutent habituellement peut offrir également un cadre d'application de contraintes strictes, afin de faciliter l'analyse statique. Le framework .NET permet par exemple d'interdire la génération dynamique de code ou l'utilisation de fonctionnalités particulières. De façon similaire, la technologie Native Client permet une exécution de code natif mais dans une forme réduite et avec une marge de manœuvre réduite, notamment en termes d'accès mémoire.

Enfin, l'analyse statique est adaptée à des programmes dont le code source est disponible, par exemple pour des scripts Web, pour des composants off-the-shelf (COTS), pour des extensions de navigateur (en Javascript), etc.

Le **premier objectif** de cette thèse est donc la formalisation de la détection de comportements suspects dans un contexte d'analyse statique ou dynamique.

Ensuite, indépendamment du contexte dynamique ou statique de l'analyse, l'analyse peut se situer à deux niveaux :

Niveau syntaxique On analyse le code dans son état brut : ses instructions, ses interactions avec le système, ses interactions réseau, etc. On cherche alors à y reconnaître un motif connu caractérisant un code malicieux. L'inconvénient est le manque de fiabilité de cette méthode : une même fonction peut être réalisée par de nombreux moyens, par le biais d'appels de librairie différents, de langages de programmation différents, etc. et il faut s'assurer que tous sont pris en compte dans la base de motifs à reconnaître. Un comportement malicieux décrit en fonction de ces interactions de bas niveau est donc peu robuste à un changement dans sa mise en œuvre.

1. Trois systèmes prédominent le marché des appareils mobiles à usage personnel : Symbian, Android et iOS. Sur les systèmes Symbian et Android, le langage Java est majoritairement utilisé et produit des exécutables dans un langage intermédiaire facilement analysable [54]. Et sur le système iOS, le langage de développement Objective-C produit des exécutables Mach-O dont le format facilite l'analyse, tandis qu'un accord de licence impose un certain nombre de contraintes aux développeurs.

Niveau sémantique On analyse la fonction réalisée par le code, en étant de ce fait indépendant de sa mise en œuvre syntaxique. On cherche alors à y reconnaître un motif connu caractérisant une fonction malicieuse. Mais la difficulté est alors triple : comment définir les motifs à reconnaître, comment déterminer la fonction réalisée par le code, et comment garantir l'efficacité de cette méthode.

Là encore, une analyse au niveau sémantique est plus puissante qu'une analyse au niveau syntaxique. En effet, l'analyse comportementale s'intéresse à l'observation d'une fonction malicieuse donnée et non à la façon dont elle est réalisée. Par exemple, il est naturel de vouloir détecter l'une de ces fonctions : lecture d'un fichier sensible, communication sur le réseau, envoi d'un ping, communication HTTP, FTP, IRC, P2P, SMB, inscription au redémarrage, accès au carnet de contacts. Peu importent les interactions sous-jacentes sur lesquelles elles reposent.

Le **second objectif** de cette thèse est donc la formalisation de l'analyse comportementale à un niveau sémantique. Nous appelons abstraction l'extraction de la fonction réalisée par une trace ou par un programme.

Dans ce travail, nous avons élaboré une méthode d'analyse sémantique du comportement du programme (i.e. de ses interactions avec le système) qui s'applique à tout contexte d'analyse, dynamique comme statique. Le fondement de notre approche est que pour identifier la fonction réalisée par le code, nous abstrayons des traces d'exécution de façon à les représenter en termes des fonctionnalités qu'elles réalisent et ainsi de s'affranchir de leurs détails d'implantation.

1.2 Travaux existants

La détection de logiciels malveillants a traditionnellement reposé sur l'utilisation de signatures binaires représentant des chaînes d'octets caractéristiques qu'il suffit de rechercher dans la représentation binaire d'un programme. Par exemple, une chaîne de caractères caractéristique d'un malware donné peut être utilisée. Un inconvénient cependant de cette méthode est sa faible robustesse devant la plus petite modification du malware, résultant en une explosion du nombre de codes malicieux distincts, au fur et à mesure que des variantes différentes d'un même malware sont disséminées pour échapper à l'utilisation d'une signature fixe. La génération de ces variantes repose sur des techniques d'auto-modification (packing, chiffrement, ...), le virus générant son code au moment de l'exécution. Mais une fois le code généré, le malware redevient identifiable par analyse de sa mémoire. Les antivirus se sont donc adaptés en

émulant le malware dans un environnement confiné et en analysant le code généré à la recherche d'une signature connue. Les auteurs de malware se sont à nouveau adaptés, de deux façons. D'une part en utilisant des techniques anti-émulation, par exemple en détectant l'émulation ou en retardant suffisamment le déchiffrement pour que l'émulateur renonce. Et d'autre part en modifiant le code du malware directement au niveau des instructions, en jouant par exemple sur la possibilité de réaliser le même calcul en utilisant des séquences d'instructions différentes : les codes résultants sont dits métamorphes.

Des techniques structurelles, plus robustes, ont été proposées par la suite, reposant sur une analyse du graphe de flot de contrôle, en partant de l'hypothèse que celui-ci est globalement peu altéré, d'une variante à une autre. Les graphes de flot de contrôle sont alors comparés par analyse sémantique [30], par isomorphisme de sous-graphe [27], par intersection d'automates d'arbres [24] ou par mesures de similarité [38, 11].

Toutefois, ces méthodes de détection permettent de détecter des codes connus ou des variantes proches de codes connus. Une autre approche s'est donc développée s'attaquant au problème de l'identification de codes inconnus. Introduite par Cohen en 1987 [33], la détection comportementale procède par analyse dynamique des interactions d'un programme avec le système, en partant du constat que l'activité malicieuse à proprement parler d'un malware s'exprime directement dans ces interactions. Cette analyse comportementale souffre néanmoins de la difficulté de caractériser de façon certaine un comportement malicieux, puisque tout comportement peut être ambivalent : envoyer mille mails à la suite est un comportement a priori suspect de la part d'un programme mais peut pourtant provenir de l'envoi d'une lettre d'information par un client de messagerie. Par ailleurs, comme évoqué précédemment, il n'est pas toujours simple de corréler les paramètres de deux interactions. Il en résulte des taux d'erreur assez élevés, comme le montre [37].

Cependant, en dépit de ces limitations, les méthodes de détection comportementale restent un outil prometteur pour découvrir des codes inconnus. Elles ont traditionnellement été formalisées par des machines d'états finis [79, 96, 20]. Dans [79], les auteurs proposent un système expert surveillant l'exécution d'un programme à la recherche d'un comportement malicieux connu. Ce comportement est décrit par un système de transitions dont les états peuvent être paramétrés par les données précédemment collectées. Forrest et al. [58] proposent d'utiliser des N-grams pour représenter des séquences d'appels système (sans paramètres) observées chez des programmes sains, un programme étant alors détecté comme malicieux si son comportement ne correspond à aucune de ces séquences. Sekar et al. [96] améliorent ensuite l'approche de Forrest et al. en représentant l'ensemble des séquences saines d'appels système par un

automate d'états finis. Bergeron et al. [20] définissent également un comportement malicieux par un automate d'états finis décrivant des séquences d'appels de librairie (sans paramètres) mais, à la différence des approches précédentes, ils se placent dans le cadre d'une analyse statique, utilisant le désassembleur IDA [5] pour calculer le graphe de flot de contrôle d'un programme et en déduire un graphe contenant uniquement les appels de librairie effectués par le programme. Dans toutes ces approches, les comportements sont définis à partir des interactions de bas-niveau observées.

Plus récemment, observant que les comportements recherchés sont en général des comportements de haut niveau, Martignoni et al. [81] proposèrent d'utiliser plusieurs couches de spécification de comportements, chaque comportement étant décrit par un graphe AND/OR dont les nœuds sont des comportements de la couche inférieure, paramétrés par les données manipulées. La couche la plus basse représente les interactions de bas-niveau observées, tandis que la couche la plus haute représente les comportements malicieux recherchés. Une trace d'un programme est alors capturée par émulation et fournie en entrée aux graphes de comportements de plus bas niveau. Quand un comportement est reconnu par un graphe, il est fourni aux graphes du niveau supérieur, jusqu'à ce que soit reconnu un comportement malicieux du niveau le plus haut. Ainsi, les spécifications de comportements malicieux ne dépendent plus de la mise en œuvre concrète du comportement. Néanmoins, ils considèrent uniquement la détection dans une trace d'un comportement malicieux et n'étudient pas de formalisme d'abstraction, i.e. de transformation de traces en représentations de haut niveau, à des fins d'analyse manuelle. Par ailleurs, ils se limitent à un scénario dynamique et ne s'intéressent pas à une application statique de l'abstraction, permettant d'étudier un ensemble de traces suffisamment exhaustif pour représenter le comportement d'un programme.

Récemment également, Jacob et al. [66] proposèrent d'abstraire des traces d'exécution collectées dynamiquement en utilisant des grammaires attribuées, dont les attributs représentent les données collectées. L'abstraction permet de transformer une trace de façon à représenter les "classes d'interaction" observées et non les appels système ou appels de librairie observés. Ainsi, l'appel `CreateFile` sera transformé en "Création d'un objet de type fichier", l'appel `send` en "Écriture de données sur un objet de type socket", etc. Un comportement est ensuite défini comme des combinaisons de ces classes d'interactions. Cette approche a plusieurs limitations. Tout d'abord, leur formalisme considère uniquement des fonctionnalités réalisées par une seule action et ne permet pas l'abstraction de fonctionnalités complexes, c'est-à-dire décrites par une séquence d'appels de librairie, d'appels système, etc. Ensuite, conséquence de la précédente limitation, les comportements sont eux-même spécifiés à un niveau

assez bas, contraint par le type de fonctionnalités abstraites. Ainsi, l'objectif de rendre la spécification de comportements malicieux plus robuste n'est que partiellement atteint puisque des réalisations de fonctionnalités complexes doivent toujours être prises en compte dans la définition du comportement et non dans les fonctionnalités de base composant le comportement. Troisièmement, l'algorithme de détection résultant de cette approche a une complexité exponentielle en temps et en espace. Enfin, là encore, l'approche ne s'applique qu'à une seule trace et exclut donc un scénario d'analyse statique.

Alternativement, Bayer et al. [17] ont étudié l'abstraction de traces d'exécution dans un contexte dynamique à des fins d'analyse automatisée. Ils définissent des comportements pour un certain nombre d'opérations élémentaires comme l'ouverture de fichier ou l'envoi de données sur le réseau et transforment alors la trace en un "profil de comportement" décrivant une séquence d'opérations élémentaires et les relations entre les paramètres de ces différentes opérations. Ils en déduisent ensuite un certain nombre de caractéristiques leur permettant de classer automatiquement le programme. Néanmoins, comme dans l'approche de Jacob et al., leur approche a deux limitations : là encore, les fonctionnalités de base ne peuvent pas représenter des séquences complexes d'interactions et leur approche ne s'applique qu'à une trace, capturée par analyse dynamique.

Avec une approche également basée sur des graphes, dans [74, 89], les auteurs représentent une trace par un graphe de comportement dont les arêtes définissent des dépendances de données. Toutefois, aucune abstraction n'a lieu et ces graphes sont directement utilisés pour identifier des comportements malicieux connus.

Enfin, dans [70, 99], les auteurs proposent de spécifier des comportements malicieux par des formules de logique temporelle, ce qui leur permet notamment d'analyser les données manipulées par les interactions avec le système. Toutefois, outre l'application uniquement dynamique, ils considèrent des spécifications de bas niveau, sans notion d'abstraction de trace.

Notre objectif est donc de proposer un modèle formel de détection de comportements malveillants par abstraction :

- qui permette d'abstraire des fonctionnalités complexes, définies au départ par des séquences d'interactions de bas niveau ;
- qui soit applicable à un ensemble quelconque de traces, en particulier à un ensemble infini de traces provenant d'une analyse statique ;
- qui fournisse un algorithme de détection, de coût raisonnable en temps et en espace.

1.3 Contributions

Nous définissons un comportement comme une séquence d'interactions d'un programme avec le système dans lequel il s'exécute. Nous considérons, en fait, des traces d'exécution restreintes à des informations particulières. En représentant les comportements à l'aide de langages formels ou à l'aide d'une algèbre de termes, nous proposons un cadre formel pour l'analyse comportementale de programmes (Chapitre 2), opérant sur une représentation de l'ensemble des traces d'exécution d'un programme (Chapitre 3).

Nous définissons alors formellement l'abstraction de comportements, i.e. la transformation d'un comportement en une représentation de plus haut niveau, permettant de s'abstraire de la mise en œuvre concrète des différentes fonctionnalités composant le comportement. Chaque fonctionnalité est représentée comme une formule de logique temporelle sur l'algèbre de termes, validée par l'ensemble des traces réalisant cette fonctionnalité. Un système de réécriture est alors associé à chaque fonctionnalité et l'abstraction des fonctionnalités est définie comme la relation de réduction associée à l'union de ces systèmes. Nous appliquons ensuite ce formalisme d'abstraction à l'analyse comportementale dynamique, c'est-à-dire à l'analyse d'un ensemble *fini* de traces collectées lors d'exécutions du programme, et à l'analyse comportementale statique, c'est-à-dire à l'analyse d'un ensemble *non borné* de traces calculé par analyse statique de code.

Nous définissons alors la notion de comportement abstrait dans ce formalisme comme une formule de logique temporelle et nous étudions les algorithmes de détection de ces comportements abstraits dans les ensembles de traces considérés, ainsi que leur complexité. En particulier, nous déduisons un algorithme linéaire en temps et en espace permettant de détecter un comportement abstrait dans un ensemble régulier, non borné, de traces. Pour cela, nous étudions successivement le problème selon une approche par réécriture de mots (Chapitre 4) puis selon une approche par réécriture de termes (Chapitre 5).

Nous généralisons ensuite ce formalisme à un formalisme d'abstraction pondérée permettant de représenter l'incertitude de l'abstraction de fonctionnalités ou de la détection d'un comportement abstrait. Pour ce faire, nous étendons le système de réécriture de termes définissant l'abstraction à un système de réécriture pondéré sur un semi-anneau commutatif. Nous étendons alors nos algorithmes de détection de comportements abstraits à la détection pondérée de comportements abstraits, en maintenant une complexité linéaire en temps et en espace (Chapitre 6).

Nous validons nos résultats par un ensemble d'expériences, menées sur des codes malicieux collectés notamment sur un pot de miel du Laboratoire de

Haute Sécurité du Loria.

Chapitre 2

Fondements de l'Analyse Comportementale

2.1 Qu'est-ce qu'un Comportement ?

En analyse comportementale, la notion de comportement se rapporte aux interactions du programme avec le système. Ces interactions peuvent s'exprimer sous différentes formes et à différents niveaux. Dans le cas de code assembleur x86, il peut s'agir des appels système effectués par le code, i.e. des interruptions x86 habituellement faites depuis les librairies système. Plus généralement, pour des codes en assembleur x86, en langage intermédiaire Java ou .NET, en langage Javascript, etc. il peut s'agir des appels aux librairies du système. Il peut également s'agir des interactions avec le réseau, soit par l'observation des communications réseau brutes, soit, au niveau des protocoles réseau, par une observation des requêtes HTTP (web), FTP (transfert de fichiers), SMTP (envoi de mails), etc. Il peut également s'agir des interactions avec le système de fichiers, ou des interactions avec les autres processus (communication inter-processus, sémaphores, etc.).

Par ailleurs, chaque action composant le comportement peut être paramétrée par les données qu'elle manipule. Dans le cas d'un appel de librairie, il s'agira des paramètres d'entrée et de la valeur de retour. Dans le cas d'une interaction réseau, il pourra s'agir du destinataire, des données envoyées et d'un code d'erreur.

En somme, un comportement représente donc une séquence d'échanges entre le programme et son environnement extérieur, qui caractérisent l'exécution du programme sur le système. Un comportement décrit donc fondamentalement une trace d'exécution particulière, composée d'actions potentiellement

paramétrées.

Dans ce document, nous considérons pour nos exemples le cas particulier où les actions sont les appels aux librairies du système (ainsi que des appels système lorsque ceux-ci sont effectués directement par le programme), ce qui nous permet d'obtenir des traces d'exécution relativement expressives. En particulier, nous privilégions les appels de librairie aux appels systèmes car, outre le fait que chaque appel de librairie se transforme en une séquence d'appels système qui sera plus délicate à interpréter, les appels de librairies sont sémantiquement plus significatifs, ne nous font pas perdre en généralité et permettent de représenter des appels à des fonctions n'exécutant aucun appel système (comme des fonctions de manipulation de chaîne, de compression, d'accès aux variables d'environnement, etc.). Cet avantage est d'ailleurs également relevé par Bayer et al. [17] à propos des données collectées par le système Anubis, un système d'analyse dynamique automatique de codes malicieux.

Ainsi, considérons la fonction suivante d'un programme en Java :

```
String readFile(String path) {
  BufferedReader in = new BufferedReader(new FileReader(path));
  String line, txt = "";
  while ((line = in.readLine()) != null) {
    txt += line + "\n";
  }
  in.close();
  return txt;
}
```

Son comportement lors d'une exécution consiste en une séquence composée d'une action `FileReader.new`, d'une action `BufferedReader.new`, d'une suite d'actions `BufferedReader.readLine` puis d'une action `BufferedReader.close`. Nous pourrions aussi paramétrer les actions et, en prenant comme convention que le premier paramètre représente la valeur de retour, observer alors le comportement consistant en une séquence composée d'une action `FileReader.new(x, path)`, d'une action `BufferedReader.new(in, x)`, d'une suite d'actions `BufferedReader.readLine(line, in)` et enfin d'une action `BufferedReader.close(_, in)`.

Une trace d'exécution étant fondamentalement une séquence d'actions, nous pouvons l'interpréter de deux manières différentes :

- Comme un mot, sur un alphabet contenant l'ensemble des actions, éventuellement paramétrées ;
- Comme un terme, sur un alphabet décomposé en un ensemble de constructeurs de séquences, un ensemble de constructeurs d'actions et un ensemble de constructeurs de données représentant les paramètres des ac-

tions.

Dans notre travail, nous considérons d'abord l'approche basée sur les mots puis, pour rendre notre formalisme plus flexible et plus puissant, nous considérerons l'approche basée sur les termes. Chaque approche a néanmoins ses avantages. L'approche sur les termes est la plus flexible car elle inclut le cas des mots comme un cas particulier et elle permet de représenter les paramètres des actions de façon naturelle. Mais l'approche sur les mots est plus simple et elle permet de faire appel à des algorithmes optimisés pour les mots.

2.2 Définitions

Langage de Mots

Soit Σ un alphabet fini. L'ensemble Σ^* dénote l'ensemble des mots finis sur Σ. Les sous-ensembles de Σ^* sont appelés des langages sur Σ. Le mot vide est noté ϵ.

Soit Σ' un alphabet fini. La projection d'un mot u de Σ^* sur Σ' est notée $u|_{\Sigma'}$ et définie de façon homomorphique, pour tout $a \in \Sigma$, par : $a|_{\Sigma'} = a$ si $a \in \Sigma'$ et $a|_{\Sigma'} = \epsilon$ sinon. La projection est étendue de façon naturelle aux langages sur Σ, la projection sur Σ' d'un langage $L \subseteq \Sigma^*$ étant notée $L|_{\Sigma'}$.

Définition 1 (Automate Fini)**.** Un *automate fini* sur un alphabet Σ est un quadruplet $A = (Q, \delta, q_0, F)$ où :
- Q est un ensemble fini d'états ;
- $\delta : Q \times \Sigma \rightarrow Q$ est une relation de transition ;
- $q_0 \in Q$ est l'état initial ;
- $F \subseteq Q$ l'ensemble des états finaux.

Une exécution de A sur le mot $w = a_0 a_1 \cdots a_n$ est une suite d'états $r = q_0 q_1 \cdots q_{m \leq n+1}$ telle que : $\forall i < m, q_{i+1} \in \delta(q_i, a_i)$. L'exécution r est réussie si $m = n + 1$ et $q_m \in F$; dans ce cas, on dit que w est reconnu par A. L'ensemble des mots pour lesquels il existe une exécution réussie de A est le langage reconnu par A et est noté $\mathcal{L}(A)$.

Les langages reconnus par un automate fini sont appelés langages réguliers de mots.

La taille d'un automate fini $A = (Q, \delta, q_0, F)$ est notée $|A|$ et est définie par : $|A| = |Q| + |\delta|$.

Algèbre de Termes

Soit un ensemble de sortes $S = \{\text{Trace}, \text{Action}, \text{Data}\}$ et une signature S-sortée $\mathcal{F} = \mathcal{F}_t \cup \mathcal{F}_a \cup \mathcal{F}_d$, où $\mathcal{F}_t$, $\mathcal{F}_a$, $\mathcal{F}_d$ sont mutuellement distincts et :

- $\mathcal{F}_t = \{\epsilon, \cdot\}$ est l'ensemble des constructeurs de traces ;
- $\mathcal{F}_a$, un ensemble fini de symboles de fonction ou de constantes, de signature $\text{Data}^n \to \text{Action}$, $n \in \mathbb{N}$, décrivant des actions.
- $\mathcal{F}_d$, un ensemble fini de constantes de type Data, décrivant des données.

On note $T(\mathcal{F}, X)$ l'ensemble des termes S-sortés sur un ensemble X de variables S-sortées. Pour une sorte $s \in S$, on note $T_s(\mathcal{F}, X)$ la restriction de $T(\mathcal{F}, X)$ aux termes de sorte s et on note X_s l'ensemble des variables de X de sorte s.

Un terme de sorte Action est appelé une action et est de la forme $f(d_1, \ldots, d_n)$ avec $f \in \mathcal{F}_a$, d'arité n, et $d_1, \ldots, d_n \in \mathcal{F}_d$. Un terme de sorte Trace est appelé une trace et est de la forme $\cdot(a_1, \cdot(a_2, \ldots \cdot(a_n, \epsilon)))$, où $a_1, \ldots, a_n$ sont des actions. On distingue la sorte Action de la sorte Trace mais par souci de lisibilité, on pourra noter a la trace $\cdot(a, \epsilon)$ pour une action a. De même, on utilise le symbole $\cdot$ de façon infixe et avec associativité à droite, et ϵ sera sous-entendu lorsqu'il n'y a pas d'ambiguité. Par exemple, si a, b et c sont des actions, la notation $a \cdot b \cdot c$ désigne la trace $\cdot(a, \cdot(b, \cdot(c, \epsilon)))$.

Automate d'Arbres

Définition 2 (Automate d'Arbres [34])**.** Un automate d'arbres (descendant) est un quadruplet $A = (\mathcal{F}, Q, q_0, \Delta)$ où $\mathcal{F}$ est un alphabet fini, Q est un ensemble fini d'états, $q_0 \in Q$ est un état initial et Δ est un ensemble fini de règles de la forme :

$$q(f(x_1, \ldots, x_n)) \to f(q_1(x_1), \ldots, q_n(x_n))$$

où $f \in \mathcal{F}$ est un symbole d'arité $n \in \mathbb{N}$, $q, q_1, \ldots, q_n \in Q$ et $x_1, \ldots, x_n$ sont des variables distinctes d'un ensemble X de variables.

La relation de transition $\to_A$ associée à A est définie par :

$$\begin{gathered}
\forall t, t' \in T(\mathcal{F} \cup Q), \\
t \to_A t' \\
\Leftrightarrow \\
\exists\, q(f(x_1, \ldots, x_n)) \to f(q_1(x_1), \ldots, q_n(x_n)) \in \Delta, \\
\exists p \in Pos(t), \exists u_1, \ldots, u_n \in T(\mathcal{F}), \\
t|_p = q(f(u_1, \ldots, u_n)) \\
\text{et } t' = t[f(q_1(u_1), \ldots, q_n(u_n))]_p.
\end{gathered}$$

Le langage reconnu par A est défini par : $\mathcal{L}(A) = \{t \mid q_0(t) \rightarrow_{\mathcal{A}}^{*} t\}$. Les langages d'arbres reconnus par des automates d'arbres (descendants) sont les langages réguliers d'arbres.

La taille de A est définie par : $|A| = |Q| + |\Delta|$.

Dans la suite, l'appellation automate d'arbres désignera spécifiquement de tels automates d'arbres descendants sans ϵ-transitions.

2.3 Ensemble de Traces d'un Programme

Traces d'Exécution

Nous introduisons maintenant un modèle de machine abstraite à partir duquel nous définissons formellement les notions de trace d'exécution et de comportement. Nous considérons des traces et des comportements finis, c'est-à-dire que lorsqu'un programme peut s'exécuter indéfiniment, nous considérons comme traces d'exécution les préfixes finis de ces exécutions infinies.

Une machine abstraite $\mathcal{M}$ consiste en un triplet $(\mu_0, IP_0, \rightarrow)$ où (μ_0, IP_0) est une configuration initiale de $\mathcal{M}$ et $\rightarrow$ est une fonction de transition de *Configurations* vers *Configurations*, où *Configurations* dénote l'ensemble des configurations de $\mathcal{M}$.

Une configuration de $\mathcal{M}$ est une paire (μ, IP) où :

- μ : *Addresses* $\rightarrow$ *Data* représente la mémoire de $\mathcal{M}$. *Addresses* est l'ensemble des adresses de $\mathcal{M}$ et *Data* est l'ensemble des valeurs ; tous deux sont des sous-ensembles de $\mathbb{N}$;
- $IP \in$ *Addresses* est le pointeur d'instruction.

Ainsi, nous avons $(\mu, IP) \rightarrow (\mu', IP')$ si la machine $\mathcal{M}$ exécute l'instruction à l'adresse IP de la mémoire μ. La mémoire μ' est la mémoire obtenue après avoir exécuté l'instruction et IP' est l'adresse de la prochaine instruction à exécuter. Un programme est un ensemble d'instructions. Une *exécution* d'une machine abstraite $\mathcal{M}$ est une séquence finie de la forme :

$$(\mu_0, IP_0) \rightarrow (\mu_1, IP_1) \rightarrow \ldots \rightarrow (\mu_n, IP_n).$$

Dans notre scénario, la configuration (μ_0, IP_0) est la configuration initiale de la machine $\mathcal{M}$. Une programme est chargé dans la mémoire μ_0 à l'adresse pointée par le pointeur d'instruction IP_0. Donc, au début d'une exécution, un programme est exécuté dans un environnement initial (également) donné par μ_0. À chaque étape, nous observons des interactions avec l'"extérieur" via la mémoire. Bien que notre modèle de machine abstraite puisse seulement

représenter des programmes à un seul thread, cette formalisation des communications est suffisante pour nos besoins.

Observons que nous nous concentrons sur les machines abstraites plutôt que les langages de programmation. Il y a plusieurs raisons à cela. Tout d'abord, notre modèle nous permet de parler de langages de programmation à n'importe quel niveau d'abstraction. De plus, dans le contexte de l'analyse de malware, les programmes sont en général auto-modifiants, autrement dit le code du programme est variable, contrairement à l'approche classique en sémantique [53]. De plus, les instructions de bas niveau, comme dans le cas du x86, ne sont pas de taille fixe et un programme peut cacher son code en rompant l'alignement des instructions. Enfin, les programmes sont traités comme de simples données et, inversement, toute suite de données peut être interprétée comme du code. Nous pensons donc que ce modèle de machine abstraite est approprié pour sous-tendre notre formalisme.

Nous formalisons tout d'abord la capture de certaines données d'une exécution de la machine $\mathcal{M}$. Ces données sont représentées à l'aide d'un alphabet Σ. Par exemple, lorsque les données capturées représentent les appels de librairies, Σ représente l'ensemble fini de ces appels.

Modélisation des Traces d'Exécution par des Mots

Définition 3 (Opérateur de capture)**.** Un *opérateur de capture* par rapport à Σ est un opérateur $\pi : Configurations \to \Sigma \cup \{\epsilon\}$ qui associe à une certaine configuration les données capturées si elles existent, et ϵ dans le cas contraire.

Notons que, dans le cas général, les données capturées pour une configuration c_n peuvent dépendre de l'historique de l'exécution, i.e. de configurations $c_{i_1} \dots c_{i_k}$ pour $i_1 \dots i_k \in [1..n-1]$. Ceci est par exemple requis pour capturer des comportements non atomiques (par exemple, une connexion SMTP est la combinaison d'une connexion réseau et de la réception d'un message de la forme "220 .* SMTP Ready"). Par souci de simplicité, nous ne considérons pas $c_{i_1} \dots c_{i_k}$ dans la définition de π.

Définition 4 (Trace d'exécution)**.** Soit $\mathcal{M}$ une machine, $e = c_1 \dots c_n$ une exécution de $\mathcal{M}$ et π un opérateur de capture par rapport à $\mathcal{F}_\Sigma$. Alors le mot $\pi(c_1) \dots \pi(c_n) \in \Sigma^*$ est la *trace d'exécution* de l'exécution e de $\mathcal{M}$ par rapport à π, notée $\pi(e)$.

Par exemple, lorsque l'on considère pour π l'opérateur capturant les appels de librairies, le mot $fopen \cdot fwrite$ représente la trace d'exécution effectuant un appel d'ouverture de fichier $fopen$ suivi d'un appel d'écriture de fichier $fwrite$.

Modélisation des Traces d'Exécution par des Termes

Lorsque l'on analyse les appels de librairies, la formalisation par des mots ne permet pas de représenter les paramètres de ces appels. Dans une seconde approche, on définit donc une trace d'exécution par un terme de l'algèbre définie en Section 2.2. Ainsi, l'alphabet Σ est désormais un sous-ensemble de l'alphabet $\mathcal{F}_a$ et les constantes de $\mathcal{F}_d$ identifient les arguments et les valeurs de retour des appels de librairies. L'autre partie de $\mathcal{F}_a$ sera constituée des symboles utilisés pour l'abstraction, comme nous le verrons plus tard. Un appel de librairie est alors représenté par un terme $f(d_1, \ldots, d_n) \in T_{\text{Action}}(\mathcal{F})$ où $f \in \Sigma$ identifie l'appel de librairie et $d_1, \ldots, d_n \in \mathcal{F}_d$ identifient les paramètres de l'appel.

Notons $T_{\text{Action}}(\mathcal{F}_\Sigma)$ la restriction de l'ensemble de termes $T_{\text{Action}}(\mathcal{F})$ au cas où $\mathcal{F}_a = \Sigma$.

Définition 5 (Opérateur de capture)**.** Un *opérateur de capture* par rapport à Σ est un opérateur $\pi : Configurations \to T_{\text{Action}}(\mathcal{F}_\Sigma) \cdot \epsilon \cup \{\epsilon\}$ qui associe à une certaine configuration les données capturées si elles existent, et ϵ dans le cas contraire.

Désormais, nous considérons plus particulièrement l'opérateur capturant les appels de librairies (ainsi que les appels système effectués directement par le programme).

Définition 6 (Trace d'exécution)**.** Soit $\mathcal{M}$ une machine, $e = c_1 \ldots c_n$ une exécution de $\mathcal{M}$ et π un opérateur de capture par rapport à $\mathcal{F}_\Sigma$. Alors le terme $\pi(c_1) \ldots \pi(c_n) \in T_{\text{Trace}}(\mathcal{F}_\Sigma)$ est la *trace d'exécution* de l'exécution e de $\mathcal{M}$ par rapport à π, notée $\pi(e)$.

Par exemple, en définissant $\mathcal{F}_d$ comme un sous-ensemble fini de $\mathbb{N}$, le terme $fopen(1,2) \cdot fwrite(1,3)$ représente la trace d'exécution de l'appel d'ouverture de fichier $fopen(1,2)$ suivi de l'appel d'écriture de fichier $fwrite(1,3)$, où $1 \in \mathcal{F}_d$ identifie le handle de fichier retourné par le premier appel, $2 \in \mathcal{F}_d$ identifie le chemin du fichier et $3 \in \mathcal{F}_d$ identifie les données écrites.

Remarque 7. Supposons qu'un appel de librairie ait une signature `f(S* s)`, où `S` est un type structure contenant deux champs `a` et `b`. Supposons maintenant que nous observions l'appel `f(s)`. Le représenter par une action $f(d_s)$ avec $d_s \in \mathcal{F}_d$ ne permet pas d'inspecter l'objet pointé par `s` et rend donc plus difficile l'analyse du flux de données. Pour résoudre ce problème, on transforme un appel `f(s)` en une action $f(d_s, d_{s_a}, d_{s_b})$, où $d_s \in \mathcal{F}_d$ identifie l'objet `s`, $d_{s_a} \in \mathcal{F}_d$ identifie l'objet `s->a` et $d_{s_b} \in \mathcal{F}_d$ identifie l'objet `s->b`.

Plus généralement, on modifie la signature des appels de librairies de façon à déréférencer les pointeurs (à un niveau comme en Java ou à plusieurs niveaux comme en C) et à compléter tout paramètre composite (i.e. représentant une classe ou une structure) par une séquence de paramètres décrivant les champs de la structure. On ignore de plus les paramètres sans intérêt pour la détection comportementale.

Notons que la représentation d'un objet de type structure par une liste d'objets représentant les champs de la structure est fréquente en model checking. C'est par exemple l'approche adoptée par le model checker Spin [59] pour représenter les structures de données. L'avantage de cette approche est que les dépendances de données mettant en jeu des champs de structures sont alors prises en compte plus facilement. On tire le même avantage du remplacement d'un pointeur par l'objet pointé.

Traces d'Exécution Étendues

En plus des appels de librairies effectuées et des relations entre leurs arguments, nous souhaitons enrichir les traces d'exécution avec des informations sur les données manipulées. En effet les constantes de $\mathcal{F}_d$, qui représentent les données, ne contiennent aucune information sur des valeurs spécifiques que peuvent prendre ces données. Par exemple, écrire des données dans un fichier est anodin, à moins que ce fichier ne soit un fichier système auquel cas on souhaite représenter cette information dans la trace d'exécution. De même, si le programme se réplique, par copie de fichier, on souhaite distinguer cette copie d'une copie quelconque en représentant, dans la trace, le fait que le fichier copié est une auto-référence. Dernier exemple, un programme peut s'inscrire dans la liste des programmes à lancer au redémarrage de l'ordinateur en indiquant son chemin dans une clé `Run` de la base de registre Windows : on souhaite donc pouvoir distinguer une écriture d'une telle clé `Run` de l'écriture d'une clé quelconque de la base de registre.

Or les arguments des appels de librairies sont des constantes de $\mathcal{F}_d$ représentant les objets manipulés par le programme sans information sur la nature de ces objets. Par exemple, supposons que le symbole $fopen \in \Sigma$, associé à la fonction d'ouverture de fichier `fopen`, prenne deux arguments : la valeur de retour et le fichier ouvert. L'action $fopen(2, 1)$ ne contient aucune information sur la nature du fichier représenté par la constante $1 \in \mathcal{F}_d$.

On adapte donc la définition des symboles de Σ de façon à leur ajouter, pour chaque propriété de l'appel que l'on souhaite représenter, un nouvel argument prenant des valeurs prédéfinies de $\mathcal{F}_d$. Ainsi, on peut ajouter un troisième argument à la fonction $fopen$ dont le profil devient alors : $\text{Data} \times \text{Data} \times \text{Data} \rightarrow$

Action. Une valeur de 0 pour cet argument indique un fichier quelconque, une valeur de 1 indique un fichier système, etc. Une ouverture de fichier système sera alors représentée par l'action $fopen(2, 1, 1)$. D'autres propriétés peuvent être représentées, comme le type du fichier ouvert (fichier de librairie `.dll`, fichier exécutable `.exe`, etc.), le type d'accès demandé (lecture, écriture, etc.).

Exemple 8. Reprenons l'exemple de l'ouverture de fichier, via un appel à la fonction Windows `CreateFile`. Nous distinguons deux propriétés d'un tel appel :

- Quel est le type du fichier ouvert ? Pour cela, nous considérons un ensemble de valeurs prédéfinies de $\mathcal{F}_d$: $\{\perp, DLL, EXE, PDF\} \subseteq \mathcal{F}_d$.
- Quel est l'emplacement du fichier ouvert ? Pour cela, nous considérons un ensemble de valeurs prédéfinies de $\mathcal{F}_d$: $\{\perp, SYS, PGM, TMP\} \subseteq \mathcal{F}_d$.

Le symbole $CreateFile \in \Sigma$ prend alors 4 arguments : la valeur de retour, le nom du fichier, la propriété "type du fichier", la propriété "emplacement du fichier".

Ainsi, un appel à `CreateFile` pourra être décrit par l'action :

$$CreateFile\,(d_{ret}, d_{path}, \perp, TMP)\,, \qquad d_{ret}, d_{path} \in \mathcal{F}_d$$

qui correspond à une ouverture d'un fichier qui n'est pas un fichier `.dll`, `.exe` ou `.pdf` et qui est un fichier temporaire.

Lors de la capture d'une trace d'exécution, ces propriétés sont facilement identifiées. Lors de la construction d'un automate de traces par analyse statique, par contre, une analyse supplémentaire est requise pour déterminer quelles propriétés sont validées.

Automate de Traces

Dans la suite, nous considérerons fixé l'opérateur de capture π et nous appellerons ensemble de traces d'une machine $\mathcal{M}$, noté $\mathit{Traces}\,(\mathcal{M})$, l'ensemble non-borné des traces d'exécution (finies) de $\mathcal{M}$ par rapport à π.

De plus, nous ne distinguerons pas les termes programme et machine, lorsqu'il n'existe pas d'ambiguité.

Lorsqu'un ensemble fini de traces est considéré, typiquement dans une approche par analyse dynamique, l'ensemble de traces associé est trivialement représenté par un automate. Mais lorsque, dans une approche par analyse statique, l'ensemble des traces d'une certaine machine $\mathcal{M}$ est considéré, cet ensemble est indécidable en général et un automate ne peut donc pas le représenter précisément. Lorsque le code du programme est connu, il est en général

non régulier du fait des appels récursifs de fonction et des branchements conditionnels permettant d'observer par exemple une séquence d'actions $a^n \cdot b^n$. On construit donc une approximation régulière de cet ensemble de traces, en limitant la profondeur de récursion et en transformant les branchements conditionnels en branchements non déterministes. La limitation de la profondeur de récursion permet de pouvoir "inliner" les appels de fonctions.

Observons que cette limitation est parfaitement justifiée dans notre cas car les comportements que nous cherchons à détecter représentent des séquences finies d'action, i.e. sans itérations ou récursion. Par exemple, il s'agira de détecter une lecture de données sensibles suivie de l'envoi de ces données sur le réseau. Par ailleurs, ce choix d'inliner les appels de fonction est celui traditionnellement opéré par les model checkers pour représenter un programme par une machine à états finis. Par exemple, le model checker Spin [59] permet de définir des fonctions interprétées en réalité comme des macros. De même, les model checkers Blast [2] et Blade [1] inlinent les appels de fonction pour construire un unique automate de flot de contrôle. L'outil Bandera réalise également l'inlining des appels de méthodes avant de construire le modèle Promela [64].

Remarque 9. Notons que certains model checkers permettent de représenter sans approximation les appels de fonction. C'est par exemple le cas de dSpin [36] et de cadp [46]. Cependant, nous nous intéressons ici à la modélisation du programme par un automate d'états fini, qui d'une part nous paraît suffisamment expressive pour nos besoins, comme on vient de le dire, et d'autre part garantira l'efficacité des algorithmes associés au formalisme d'abstraction de comportements que nous présentons ici.

La limitation de la récursion induit une sous-approximation, i.e. l'automate résultant est incomplet, ce qui peut créer des faux négatifs lors de la détection. Il y a également sous-approximation lorsqu'une partie du code du programme n'est pas représentée (volontairement, si le code est jugé sans intérêt, ou involontairement, si le code n'est tout simplement jamais découvert). Ainsi, formellement, si l'ensemble $\mathit{Traces}\,(\mathcal{M})$ des traces de la machine $\mathcal{M}$ est sous-approximé par L, alors : $L \subseteq \mathit{Traces}\,(\mathcal{M})$.

De même, la transformation des branchements conditionnels en branchements non déterministes induit une sur-approximation, i.e. l'automate résultant contient des traces injustifiées, ce qui peut créer des faux positifs lors de la détection. Il y a également sur-approximation lorsque l'analyse statique produit des résultats erronés comme par exemple lorsque les arguments de deux appels de librairies sont incorrectement corrélés. Ainsi, formellement, si l'ensemble $\mathit{Traces}\,(\mathcal{M})$ des traces de la machine $\mathcal{M}$ est sur-approximé par L, alors : $\mathit{Traces}\,(\mathcal{M}) \subseteq L$.

Ainsi, un automate de traces d'une machine $\mathcal{M}$ représente une modélisation de son ensemble de traces par un automate fini. Lorsque les traces sont représentées par des mots sur Σ, on a la définition suivante.

Définition 10 (Automate de traces sur Σ)**.** Soit une machine $\mathcal{M}$ avec un ensemble de traces $\mathit{Traces}\,(\mathcal{M})$. Un *automate de traces* pour $\mathcal{M}$ par rapport à Σ est un automate fini A sur Σ tel que :

$$\exists S \subseteq \Sigma^*, S \subseteq \mathit{Traces}\,(\mathcal{M}) \wedge S \subseteq \mathcal{L}\,(A)\,.$$

Lorsque les traces d'exécution sont représentées par des termes de $T_{\text{Trace}}\,(\mathcal{F}_\Sigma)$, un automate de traces est défini de la même manière comme un automate d'arbres sur $\mathcal{F}_\Sigma$.

Définition 11 (Automate de traces sur $\mathcal{F}_\Sigma$)**.** Soit une machine $\mathcal{M}$ avec un ensemble de traces $\mathit{Traces}\,(\mathcal{M})$. Un *automate de traces* pour $\mathcal{M}$ par rapport à Σ est un automate d'arbres A sur $\mathcal{F}_\Sigma$ tel que :

$$\exists S \subseteq T_{\text{Trace}}\,(\mathcal{F}_\Sigma)\,, S \subseteq \mathit{Traces}\,(\mathcal{M}) \wedge S \subseteq \mathcal{L}\,(A)\,.$$

En fait, les alphabets $\mathcal{F}_a$ et $\mathcal{F}_d$, décrivant respectivement les symboles d'actions et les données, étant finis, l'ensemble $T_{\text{Action}}\,(\mathcal{F})$ des termes de sorte Action est lui aussi fini. On peut donc définir une transformation isomorphique simple entre des traces définies par des termes et des traces définies par des mots, ce qui permet d'adapter l'ensemble nos résultats sur les mots à des termes et vice versa, et de pouvoir faire appel indifféremment à des résultats sur les langages de mots ou sur les ensembles de termes.

2.4 Formalisation de la Détection Comportementale Classique

Avant d'aborder l'analyse comportementale par abstraction, nous étudions le problème de la détection comportementale classique dans notre formalisme, c'est-à-dire le problème de détecter dans un programme un comportement spécifique défini sur Σ. Cette formalisation a été présentée dans [18] pour des traces représentées par des mots. Nous la généralisons ici à des traces représentées par des termes, le cas des mots étant un cas particulier. Comme nous allons le voir, notre formalisme se révèle puissant à trois égards :

- Il s'applique aussi bien à un scénario d'analyse statique qu'à un scénario d'analyse à l'exécution (i.e. le programme est représenté par une trace d'exécution se construisant progressivement) ;

- Il s'applique à des comportements définis comme des formules de logique temporelle FOLTL, comme étudié dans [70, 99] ;
- Et lorsqu'un comportement représente un langage *régulier* de $T_{\text{Trace}}(\mathcal{F}_\Sigma)$, ce qui est toujours le cas en pratique, sa détection au sein d'un programme prend un temps et un espace linéaire en la taille du programme et en la taille de l'automate représentant le comportement.

Plus précisément, nous définissons un comportement sur l'alphabet Σ comme l'ensemble des traces d'exécution sur Σ qui le réalisent. La détection comportementale classique pour ce comportement consiste alors à rechercher dans les traces d'exécution d'un programme une trace contenant une occurrence de ce comportement.

Définition 12 (Comportement sur Σ)**.** Un *comportement* sur Σ est un ensemble de traces $M \subseteq T_{\text{Trace}}(\mathcal{F}_\Sigma)$.

Le comportement M est défini par une formule FOLTL close φ_M sur $AP = T_{\text{Action}}(\mathcal{F}_\Sigma, X)$ ssi :

$$M = \{t \in T_{\text{Trace}}(\mathcal{F}_\Sigma) \mid t \models \varphi_M\}.$$

Cette définition englobe en particulier les deux cas suivants :

- Un comportement sur Σ peut être défini comme l'ensemble des traces d'un code malveillant connu. Autrement dit, l'analyse d'un code malveillant donné permet de déduire l'ensemble de ses traces d'exécution malicieuses et de définir à partir de cet ensemble une signature propre au code malveillant. La détection est alors réactive, en ce sens qu'elle permet de se protéger contre des menaces existantes.
- Un comportement sur Σ peut être défini de façon générique comme l'ensemble des traces validant une certaine formule de logique temporelle FOLTL. Autrement dit, cette formule décrit des combinaisons malicieuses d'actions, sans que ces combinaisons aient été préalablement observées dans un code malicieux existant. La détection est alors proactive, en ce sens qu'elle permet de se protéger contre des menaces futures. La logique FOLTL sera décrite plus en détail dans le Chapitre 5.

Exemple 13. Le comportement sur Σ décrivant l'ensemble des traces d'exécution utilisant l'API Windows pour écrire dans un fichier système est défini par la formule FOLTL suivante :

$$\varphi := \exists x, y, z.\, CreateFile(x, y, \bot, SYS) \wedge \neg CloseHandle(x)\ \mathbf{U}\ WriteFile(x, z).$$

Notez que nous utilisons ici des traces d'exécution étendues afin de représenter des propriétés des appels de librairies étendus, comme cela a été décrit

en Section 2.3. L'appel `CreateFile` est étendu par les deux propriétés définies dans l'Exemple 8.

Une trace d'exécution valide cette formule si elle commence par une action `CreateFile` sur un fichier système suivie d'une action `WriteFile` sur le fichier ouvert, sans que l'identifiant du fichier ait été libéré entre temps.

Le problème de la détection d'un comportement M donné consiste alors à déterminer si une des traces d'exécution d'un programme exhibe le comportement M.

Définition 14. Un programme p avec un ensemble de traces L *exhibe un comportement* M sur Σ, ce qui est noté $L \Cap M$, ssi :

$$L \cap M \neq \emptyset.$$

Autrement dit, un programme p exhibe un comportement M sur Σ ssi l'une de ses traces est élément de M. Lorsque M est défini par une formule de logique temporelle φ_M, le problème de la détection se résume alors à un problème classique de model checking :

$$L \Cap M \Leftrightarrow \exists t \in L,\, t \models \varphi_M.$$

Si L est représenté par un automate d'arbres, le problème de la détection a alors une complexité linéaire en la taille de l'automate représentant L et en la taille de la formule φ_M [82].

Dans le cas général, lorsque l'ensemble de traces L est reconnu par un automate A et que le comportement M est régulier et reconnu par un automate A_M, le problème de la détection devient un test du vide de l'intersection de deux automates :

$$L \Cap M \Leftrightarrow \mathcal{L}(A) \cap \mathcal{L}(A_M) \neq \emptyset.$$

Dans la pratique, la détection s'effectue par rapport à une base de signatures de comportements réguliers sur Σ. Il suffit donc de considérer pour A_M l'union des automates de chaque comportement. On a alors le théorème suivant.

Théorème 15. *Soit $\mathcal{D}$ un ensemble de comportements sur Σ reconnus par un automate $A_{\mathcal{D}}$. Il existe une procédure décidant, pour tout programme p associé à un automate de traces A, si p exhibe $\mathcal{D}$, en temps et espace $O(|A| \times |A_{\mathcal{D}}|)$.*

Démonstration. Décider si p exhibe $\mathcal{D}$ revient à construire l'automate A' reconnaissant l'intersection de $\mathcal{L}(A)$ et $\mathcal{L}(A_{\mathcal{D}})$, ce qui prend un temps et espace $O(|A| \times |A_{\mathcal{D}}|)$ et produit un automate de taille $O(|A| \times |A_{\mathcal{D}}|)$ [34, p.31], puis

à décider si A' reconnaît l'ensemble vide, ce qui est linéaire en temps et espace en la taille de A' [34, Théorème 1.7.4].

□

Ce théorème contient comme cas particulier l'analyse d'un programme à l'exécution, puisqu'il suffit de considérer l'automate qui reconnaît la seule trace capturée. On peut donc en déduire une procédure de détection incrémentale dans un scénario de détection à l'exécution. Autrement dit, si la trace est construite progressivement, la procédure de détection peut être exécutée de façon continue, en complétant l'automate intersection reconnaissant $\mathcal{L}(A) \cap \mathcal{L}(A_{\mathcal{D}})$ au fur et à mesure que l'automate A se construit. On peut aussi directement simuler la reconnaissance dans $A_{\mathcal{D}}$ de la trace capturée en déroulant un chemin dans $A_{\mathcal{D}}$ en parallèle avec la trace.

Théorème 16. *Soit $\mathcal{D}$ un ensemble de comportements sur Σ reconnus par un automate $A_{\mathcal{D}}$. Il existe une procédure incrémentale décidant, pour chaque nouvelle action d'une trace $t \in T_{\text{Trace}}(\mathcal{F}_{\Sigma})$ construite incrémentalement, si la nouvelle trace exhibe $\mathcal{D}$, en temps et espace $O(|A_{\mathcal{D}}|)$.*

Démonstration. Soit $t' = a_1 \cdots a_n$ la trace t à l'étape n.

La procédure de détection simule la reconnaissance incrémentale de la trace t par l'automate $A_{\mathcal{D}}$ et calcule, à l'étape $n \in \mathbb{N}$, l'ensemble des états $Q_n \subseteq Q'$ où a été reconnu le sous-terme ϵ de t'.

- Initialement, $Q_0 = Q'_i$.
- À l'étape $n+1$, une action a_{n+1} est ajoutée à la trace t' et Q_{n+1} est construit de la façon suivante :
  - Initialement, $Q_{n+1} = \emptyset$;
  - Pour chaque règle $q(\cdot(x, x')) \to \cdot(q'(x), q''(x'))$ dans Δ', si $q'(a_{n+1}) \to^*_{\Delta'} a_{n+1}$ et $q \in Q_n$, alors ajouter q'' à Q_{n+1}.

Il existe au plus $|\Delta'| \in O(|A_{\mathcal{D}}|)$ règles $q'(\cdot(x, x')) \to \cdot(q(x), q''(x'))$ dans Δ' donc le calcul de Q_{n+1} prend un temps et espace $O(|A_{\mathcal{D}}|)$.

De plus, à l'étape $n \in \mathbb{N}$, la trace $t' = a_1 \cdots a_n$ exhibe $\mathcal{D}$ ssi il existe $q'' \in Q_n$ tel que $q_{n+1}(\epsilon) \to_{\Delta'} \epsilon$.

□

L'approche proposée permet donc de détecter des comportements exprimés en termes concrets, c'est-à-dire de bas niveau. Son inconvénient majeur est qu'il suffit d'utiliser des appels de librairies différents afin de réaliser une fonctionnalité similaire pour compromettre la détection du comportement. L'objectif de notre travail est donc de proposer un formalisme permettant d'effectuer la détection au niveau des fonctionnalités réalisées et non des actions concrètes

observées, autrement dit permettant de détecter un comportement de haut niveau dans un ensemble (potentiellement non borné) de traces de bas niveau.

Chapitre 3

Construction d'un Automate de Traces

Un automate de traces d'un programme peut être construit par analyse dynamique, en utilisant un ensemble de traces capturées, ou par analyse statique, en reconstruisant le code machine du programme depuis sa représentation binaire. L'approche dynamique est plus simple et plus fiable mais moins exhaustive que l'approche statique. En particulier, l'approche statique est appropriée pour des programmes écrits dans des langages de haut niveau ou pour des programmes quelconques dont les protections (notamment le packing) peuvent être désactivées en émulant le programme pendant une courte durée.

3.1 Construction par Analyse Dynamique

Dans une approche dynamique, chaque trace d'exécution correspond à un chemin dans le graphe de flot de contrôle (CFG) du programme. Les nœuds du CFG sont les différentes instructions du programme (chaque nœud étant caractérisé par l'adresse de l'instruction) et les arcs du CFG sont des transitions entre ces instructions. Chaque trace capturée, constituée d'une séquence d'instructions avec leurs adresses, peut donc être utilisée pour révéler progressivement des nœuds et des arcs de ce graphe. L'avantage d'une telle approche est que le code mort (i.e. jamais exécuté) ou les chemins impossibles ne sont pas représentés dans le CFG ainsi construit.

Dans notre cas, nous ne nous intéressons pas à toutes les instructions mais plus spécifiquement aux séquences d'appels de librairies effectués et les traces capturées ne décrivent que ces appels et leurs arguments. Aussi, plutôt que de construire un CFG complet puis d'en déduire un automate de traces en

restreignant le CFG aux instructions effectuant des appels de librairies, on construit directement un automate de traces : les états de l'automate sont les instructions du programme réalisant un appel de librairie et, comme pour le CFG, une trace capturée révèle alors un chemin dans l'automate de traces.

Dans le cadre de nos expérimentations, nous avons donc développé un outil construisant un automate de traces à partir d'un ensemble de traces d'exécution. On capture des traces d'exécution par instrumentation dynamique du code du programme à analyser, afin d'avoir un contrôle fin sur l'exécution du programme et ainsi de pouvoir accéder à des informations détaillées pour chaque appel de librairie, parmi lesquelles : appel de librairie effectué, arguments de cet appel et adresse de l'instruction à l'origine de l'appel. Seuls les appels effectués depuis le code du programme sont enregistrés ; les appels effectués depuis le code d'une librairie sont ignorés : cela permet de ne capturer le comportement du programme qu'au niveau de son code source. Par ailleurs, lors de la capture, les arguments des appels sont analysés en fonction du type du paramètre attendu. Par exemple, si le paramètre est un pointeur vers une structure de données, à la fois la valeur du pointeur et la structure pointée sont analysées. En effet, comme évoqué dans le chapitre précédent, les paramètres des appels de librairies sont développés afin de suivre plus précisément le flux de données.

Notre outil de collecte de traces repose sur Pin [8], un outil d'instrumentation dynamique de programmes. Notons que d'autres outils d'instrumentation, comme Dynamorio [4], auraient pu être utilisés, avec des résultats similaires. Notons également qu'il existe des outils permettant de capturer des traces d'exécution mais soit ils s'intéressent uniquement aux appels système (comme StraceNT), soit les informations collectées ne sont pas suffisamment détaillées pour construire l'automate de traces et son flux de données.

Les traces observées permettent ensuite de construire un automate de traces, comme expliqué précédemment. Chaque adresse d'instruction à l'origine d'un appel est associée à un état de l'automate et la trace capturée permet de compléter l'automate en "activant" les transitions empruntées par la trace.

En outre, notre outil prend en charge les threads, les processus enfants et les hooks Windows (code utilisateur que le système doit appeler lorsqu'un évènement spécifié se produit, par exemple lors d'une entrée au clavier). Une approche simple pour construire l'automate de traces résultant serait de construire, outre l'automate de traces du programme principal, l'automate de traces de chaque thread, processus ou hook puis d'autoriser un entrelacement quelconque de ces automates. Mais d'une part la taille de l'automate résultant serait prohibitive et d'autre part des chemins sans équivalent concret deviendraient possibles, rendant l'automate de traces moins précis. Par conséquent,

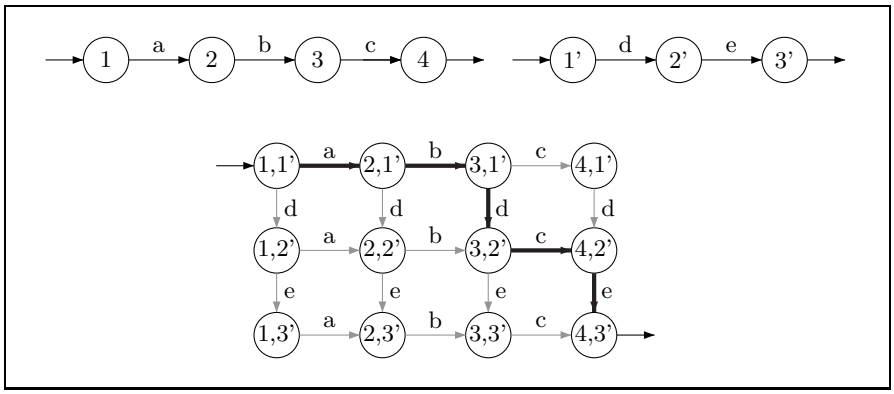

FIGURE 3.1: Exemple de construction d'un automate de traces en présence de threads.

nous adoptons une approche similaire au cas simple, sans threads, processus ou hooks : on révèle progressivement les entrelacements effectifs dans l'automate représentant tous les entrelacements possibles des threads, processus et hooks du programme. La seule différence avec le cas simple est qu'un nœud de l'automate n'est plus associé à une adresse d'instruction mais à un ensemble d'adresses d'instructions, représentant les valeurs des pointeurs d'instruction du programme principal et de chaque thread, processus ou hook. Une trace d'exécution représentant un entrelacement particulier, elle permet alors d'activer un chemin spécifique dans l'automate. Ainsi, seuls les entrelacements effectivement observés sont représentés.

Par exemple, supposons qu'un programme contienne un thread principal exécutant trois actions `a`, `b` et `c` et un deuxième thread exécutant deux actions `d` et `e`. Supposons maintenant que l'action `b` lance le deuxième thread et que l'on observe ainsi la trace d'exécution `abdce`. La Figure 3.1 illustre la construction résultante de l'automate de traces.

Notons que cette construction est similaire, dans son algorithme, à la construction proposée par Sekar et al. [96], avec la différence, toutefois, que nous capturons les appels de librairies, que nous procédons par instrumentation du code et que nous construisons également le flux de données sous-jacent. Sekar et al. capturent les appels systèmes (par nature moins expressifs que les appels de librairie), remontent la pile du processeur afin d'identifier l'instruction du programme à l'origine de l'appel et ignorent les arguments des appels système.

3.2 Construction par Analyse Statique

Dans le cas d'une approche statique, l'automate de traces est construit selon la procédure suivante :

$$\text{Programme} \Longrightarrow \begin{array}{c}\text{Code du}\\ \text{programme}\end{array} \Longrightarrow \begin{array}{c}\text{Machine à}\\ \text{états finis}\end{array} \Longrightarrow \begin{array}{c}\text{Automate}\\ \text{de traces}\end{array}$$

1. Le programme est désassemblé et son code dans le langage source ou dans un langage intermédiaire de plus haut niveau est construit (Section 3.2).
2. Une représentation par une machine à états finis est construite (Section 3.2).
3. Finalement, un automate de traces est construit (Section 3.2).

Décompilation du Code

Dans le cas de programmes dans des langages intermédiaires comme ceux des technologies .NET et Java, particulièrement présentes sur les appareils mobiles Windows Phone et Android respectivement, le code est facilement décompilé du fait de la spécification très restrictive de ces langages : des outils de décompilation existent d'ailleurs, permettant d'obtenir un code source quasiment identique au code source initial.

De même, dans le cas d'un script (en Javascript, VBScript, etc.), on a facilement accès au code source. En outre, les techniques de protection éventuellement mises en œuvre sont souvent facilement contournables. Par exemple, un script Javascript malicieux se protège typiquement en déchiffrant son code au moment de l'exécution puis en l'évaluant avec la fonction `eval`. Le code malicieux en clair s'obtient alors en capturant le premier appel à la fonction `eval`.

Enfin, dans le cas de programmes x86, le code est désassemblé par analyse statique, en suivant le flot de contrôle [31, 103], comme le fait l'outil de désassemblage IDA. C'est l'approche adoptée par Bergeron et al. [20], Christodorescu et al. [30], Singh et Lakhotia [99] ou encore Kirda et al. [72]. Lorsque le programme est protégé par des techniques de packing, des techniques d'unpacking sont utilisées, consistant à laisser le programme s'unpacker avant de l'analyser [92, 90, 16]. Notons que dans certains scénarios d'attaques ciblées, contrairement aux codes malicieux usuels, le code est peu ou pas protégé, afin de ne pas éveiller les soupçons d'un antivirus [80]. De plus, lors d'une analyse comportementale au cours d'une exécution, on peut ne commencer à analyser un programme packé que lors de sa première opération critique, opération après laquelle le programme s'est en général intégralement unpacké.

Un problème de l'analyse statique de programmes x86 réside dans l'existence de sauts et d'appels indirects : certaines parties du code du programme et certains chemins dans le code peuvent alors ne pas être découverts. Toutefois, certaines techniques d'analyse statique permettent de calculer les valeurs des adresses de saut ou d'appels [12, 71, 43]. Des techniques d'exécution symbolique permettent également de calculer ces valeurs ou de construire automatiquement de nouvelles entrées afin de découvrir de nouveaux chemins du code. C'est le cas des outils Dart [50], Sage [51] et BitScope [26] et d'un outil de Kruegel et Kirda [88], ainsi que d'une approche de McMillan [85]. Ces différents outils ont pour objectif d'extraire une vue plus exhaustive d'un programme lorsque son exécution dépend de l'environnement, en interprétant les conditions dirigeant le flot de contrôle.

Modélisation par une Machine à États Finis

On modélise ensuite le programme par une machine à états finis, dont les états correspondent aux états du programme à l'exécution. Les données considérées pour caractériser un état du programme dépendent des propriétés à vérifier sur le modèle. Par exemple, un état peut être caractérisé par le pointeur d'instruction, l'état de la mémoire (ou des variables du programme), etc. De plus, les transitions entre états peuvent être étiquetées, par exemple par l'instruction exécutée ou, lors d'un branchement conditionnel, par la condition vérifiée.

La construction d'un modèle fini exact (correct et complet) repose alors sur la finitude de l'ensemble des états du programme, finitude qui n'est pas garantie en général du fait des comportements dynamiques du programme comme l'allocation non-bornée de mémoire, la récursion non-bornée de fonctions ou la création dynamique de threads. Et lorsqu'une représentation exacte du programme par une machine à états finis existe, la taille du modèle est souvent prohibitive (par exemple, lorsque doit être représentée l'entrelacement de 10 threads complexes). Une simplification est alors cruciale et habituellement pratiquée [35, 39], l'objectif étant d'obtenir une représentation compacte d'un système en éliminant les informations superflues qui ne seront pas utilisées lors de la vérification d'une propriété sur le modèle et en approximant les données ou comportements dynamiques de façon à assurer la décidabilité et l'efficacité de cette vérification.

Ces approximations sont encore plus pertinentes en analyse comportementale. Ainsi, il semble inutile de considérer de la récursion non-bornée, de la création de threads non-bornée, des allocations de mémoire non-bornées, etc. car les comportements que nous cherchons représentent en général une sé-

quence finie de fonctionnalités. Par exemple, une fuite d'informations peut être modélisée par une capture de données suivi de l'envoi de ces données vers un emplacement extérieur : les simplifications précédentes ne compromettent pas l'observation de ces deux fonctionnalités. On décide donc plus généralement de limiter tous les paramètres dynamiques à un seuil fixé et de travailler sur le modèle fini associé. Notons que le choix de ce seuil (pour chaque paramètre dynamique) peut être affiné en fonction du comportement recherché. Par exemple, plutôt que d'interdire la récursion, on peut l'autoriser jusqu'à une profondeur donnée.

Dans ce chapitre, on s'intéresse plus particulièrement à des programmes Java et C. La construction d'un modèle fini du programme est réalisée à l'aide d'outils existants, qui permettent de modéliser un programme dans un langage de modélisation comme Promela (utilisé par Spin [59]) ou Bir (utilisé par Bandera [35, 95]). Les fondements d'une telle transformation pour des programmes Java sont étudiés dans [101] et [63].

Dans le cas de programmes Java, les outils suivants permettent de construire un modèle Promela : Bandera [64], qui génère un modèle intermédiaire en Bir, et Java Path Finder [57, 6] (dans sa première version, puisqu'il s'agit désormais d'un outil de vérification dynamique). Dans le cas de programmes C, les outils Blast [22, 2] et C to Promela [67] permettent par exemple d'obtenir un modèle Promela.

Dans la suite du chapitre, nous considérons plus spécifiquement le langage de modélisation Promela, qui est le langage utilisé par ces différents outils.

Notons que ces outils appliquent diverses techniques d'optimisation et de simplification :

- Program slicing (élimination du code n'affectant pas la vérification) [61, 56, 39] ;
- Abstraction [15] ;
- Spécialisation pour un environnement d'exécution particulier [55] ;
- Limitation de la profondeur de récursion et inlining de fonctions [32, 3] ;
- Limitation du nombre de threads ;
- Identification des mécanismes de synchronisation dans le code et représentation par des constructions spécifiques du langage Promela ;
- Partial Order Reduction [49, 40] et minimisation, pour représenter efficacement l'entrelacement de threads (source d'une explosion du nombre d'états).
- Identification des opérations locales à un thread, mise en œuvre dans l'outil Indus [39] : ces actions peuvent alors être exécutées de façon atomique et contribuer ainsi à réduire le nombre d'états induit par l'entre-

lacement de threads.

Enfin, dans le cas de programmes C, les outils BLAST et C TO PROMELA attendent en entrée un code C, qui doit donc être inféré du code désassemblé du programme, obtenu à l'étape précédente. Dans notre cas, ce code n'a pas besoin de représenter précisément le programme mais seulement ses séquences d'appels de librairies et leurs paramètres : les calculs intermédiaires peuvent donc être ignorés. Or, lorsque le programme n'est pas obfusqué [1], les appels de librairies sont facilement identifiables dans le code désassemblé, IDA les reconnaissant automatiquement. De plus, les appels de librairies attendent des paramètres de types fixés, ce qui permet de déduire partiellement la structure de la mémoire du programme. C'est en fait la technique appliquée par IDA pour interpréter la pile d'une fonction de façon intelligible. Les variables restant inconnues sont typiquement celles qui ne sont pas liées directement aux paramètres des appels de librairies et qui n'entrent donc pas en compte dans les traces représentées. Ainsi, on peut construire une approximation du code C du programme, permettant de générer un modèle PROMELA à partir de l'un des outils précédents.

Construction de l'Automate de Traces

Lorsque l'on travaille sur des mots (autrement dit, lorsque l'on ignore les arguments des appels de librairies), on construit l'automate de traces directement depuis le modèle PROMELA généré, en utilisant SPIN qui permet de construire l'automate fini représentant le modèle [2] : les transitions sont étiquetées par des instructions, ou par des conditions dans le cas de branchements conditionnels. Il suffit alors, pour nos besoins, de restreindre l'automate aux instructions représentant des appels de librairies et de remplacer par des ϵ-transitions les transitions étiquetées par des conditions (ce qui consiste à transformer les branchements conditionnels en branchements non déterministes).

Lorsque par contre on souhaite représenter les arguments des appels de librairies (i.e. on travaille sur des termes), ces arguments sont représentés sur l'alphabet des données $\mathcal{F}_d$ et le problème est alors de déterminer à quelles constantes de $\mathcal{F}_d$ associer les arguments. Nous avons, dans ce cas, besoin d'une procédure permettant d'associer de manière automatique un argument d'un appel de librairie à un symbole de $\mathcal{F}_d$. Nous nommons cette procédure abstraction du flux de données.

1. Rappelons que bien que les programmes malicieux soient packés la plupart du temps, on suppose que l'analyse statique est effectuée une fois le code unpacké en mémoire.
2. Commande `pan -D`.

Une solution intuitive pour l'abstraction du flux de données est d'indexer les variables du programme sur l'alphabet $\mathcal{F}_d$. En effet, le modèle PROMELA est obtenu par analyse statique, en associant à chaque variable du programme une variable dans le modèle ; la transformation d'un argument d'un appel de librairie dans le modèle en un symbole de $\mathcal{F}_d$ est donc immédiate.

Mais cette approche a un défaut car les dépendances dans le flux de données doivent alors être traitées de façon explicite : si deux appels de librairies se font sur des variables différentes, cela ne garantit pas pour autant que ces variables représentent des objets distincts. Par exemple, dans le code suivant, il faudra tenir compte du fait que les actions `load_data` et `use_data`, bien qu'opérant sur des variables différentes, utilisent en réalité le même objet :

```
void *a, **b;
a = malloc(1024);
b = &a;
/* load data in a */
load_data(a, 1024);
/* use data in b */
use_data(*b, 1024);
```

Plus généralement, le problème de déterminer si deux variables référencent le même objet, appelé dans la littérature problème d'analyse des aliases de pointeurs, est NP-complet lorsqu'un programme utilise une profondeur non bornée d'indirections de pointeurs [78], comme on peut le rencontrer en C.

Pour s'affranchir de l'analyse coûteuse des pointeurs lors de l'abstraction et de la détection de comportements, nous représentons par des éléments de $\mathcal{F}_d$ les objets manipulés par le programme, de telle sorte que deux appels de librairies utilisent le même objet ssi ils sont paramétrés par la même constante de $\mathcal{F}_d$. Les traces obtenues par analyse statique opérant sur les variables du programme, il faut alors les transformer de façon à exprimer leurs paramètres sur $\mathcal{F}_d$.

Pour ce faire, nous construisons une représentation abstraite de la mémoire du programme, qui est une approximation de la forme de la mémoire du programme, composée de sa pile (variables locales) et de son tas (variables dynamiques).

Construction d'une représentation abstraite de la mémoire du programme Nous utilisons des techniques d'analyse de forme [69, 60, 28, 94, 68] qui consistent à modéliser la mémoire du programme par un graphe, dont les nœuds représentent des positions dans la mémoire et les arcs représentent des références de pointeurs entre deux positions de la mémoire. Par exemple, les

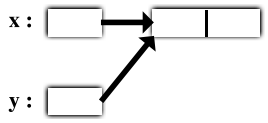

FIGURE 3.2: Exemple d'état de la représentation abstraite de la mémoire.

model checkers BLAST [23, 2] et SLAM2 [13, 14] utilisent l'analyse de forme pour traiter avec plus de précision les pointeurs et les structures de données récursives.

Des techniques existantes sont donc appliquées afin de construire une représentation abstraite de la mémoire, dont les cases mémoire seront indexées par $\mathcal{F}_d$ et telle qu'en tout point du programme, on puisse déterminer quel ensemble de cases mémoire désigne une variable.

La construction de ce type de représentation abstraite est étudiée dans [60, 68, 94] et repose sur l'abstraction du flux de données selon une méthode conservatrice, permettant de calculer, en chaque point du programme, l'ensemble des états possibles de la mémoire du programme, avec les dépendances de données sous-jacentes. Une telle abstraction est courante en analyse du flux de données (cf. Aho et Ullman [9, Chapitre 9]). Dans [60, 68, 94], un état de la mémoire est un ensemble de cases mémoire tel que chaque variable du programme est représentée sur ces cases et tel que certaines cases référencent d'autres cases. Lorsque l'allocation dynamique de mémoire est prise en compte, l'espace des états est potentiellement non borné et une approximation appelée k-limiting est appliquée, consistant à limiter à une profondeur k la profondeur des structures de données récursives.

Exemple 17. Pour l'exemple de programme suivant, avant l'exécution de la dernière instruction, l'algorithme d'Horwitz, Pfeiffer et Reps [60] construit l'état de la mémoire représenté en Figure 3.2.

```
struct LinkedList {
  int hd;
  LinkedList *tl;
}

x := new LinkedList(0, NULL);
y := x;
y := y.hd;
```

Une instruction d'allocation dynamique (`malloc`, `new`, etc.), lorsqu'elle peut être exécutée plus d'une fois, peut allouer plusieurs objets et la représentation

abstraite de la mémoire du programme doit donc a priori représenter l'ensemble de ces objets. On décide de borner par 1 le nombre de ces objets, autrement dit une telle instruction ne peut être exécutée plusieurs fois qu'à la condition que l'objet précédemment créé ait été libéré entre temps (par `free`, `delete`, etc.). Cela nous permet de considérer que la représentation abstraite de la mémoire du programme reste la même au cours de l'exécution, c'est-à-dire qu'entre deux états de la mémoire, seules varient l'application associant une variable à un ensemble de cases mémoire et les dépendances de données entre ces cases mémoire.

Ainsi, une représentation abstraite de la mémoire d'un programme est définie de façon unique à partir de l'ensemble des objets du programmes, qui se décomposent en :

- les objets locaux, i.e. alloués statiquement (dans la pile) : `x` et `y` dans l'Exemple 17 ;
- les objets alloués dynamiquement (dans le tas) : l'objet alloué par `new LinkedList(0, NULL)` dans l'Exemple 17.

De plus, on tient compte des constantes (nombres, chaînes de caractères, etc.) utilisées comme paramètres des appels de librairies en les assimilant à des variables du programme, autrement dit on les représente explicitement dans la représentation abstraite de la mémoire.

Pour terminer, notons qu'on peut approximer certaines structures de données afin de simplifier la représentation abstraite de la mémoire du programme (et par extension l'analyse du flux de données lors de l'analyse comportementale) :

- On traite les collections (tableaux, listes, etc.) comme des ensembles prédéfinis d'éléments associés à des variables distinctes, ce qui est une approche usuelle (par exemple, le model checker Spin simule des tableaux de taille fixe en les remplaçant le moment venu par un ensemble fini de variables). On peut même encore simplifier le modèle, comme dans [93], en considérant qu'une collection contient au plus un élément car ses différents éléments ont en général un rôle similaire (par exemple la collection contient un ensemble de fichiers à infecter, un ensemble d'adresses mail destinataires d'un spam, etc.).[3]
- Dans le cadre de l'analyse comportementale, certains champs des objets ne nous intéressent jamais et une flexibilité peut être introduite en altérant les structures de données, de façon à ce que certains champs soient

3. Notons que lorsqu'on représente une collection par un ensemble prédéfini d'éléments, il faut alors simuler les champs associés à la collection (par exemple le champ `length` d'un tableau) par des constantes ou par des accesseurs `get` et `set`.

ignorés ou ne deviennent accessibles que par des pseudo-accesseurs `get` ou `set`.

L'exemple suivant illustre sur un cas concret la construction d'une représentation abstraite de la mémoire d'un programme.

Exemple 18. Considérons le code suivant extrait d'une application mobile réelle, SMS_Replicator_Secret, pour systèmes Android, qui fait suivre vers un numéro tiers tous les SMS reçus ou envoyés. Cette application définit une classe `SMSReceiver` avec une méthode particulière `onReceive(Context context, Intent intent)` et demande à Android, via son fichier de métadonnées, d'exécuter cette méthode à chaque SMS reçu ou envoyé. Lorsqu'un SMS est reçu ou envoyé, le système exécute alors cette méthode en initialisant le champ `extras[‘‘pdus’’]` du paramètre `intent` par l'objet représentant le SMS concerné.

Le code ci-dessous a été obtenu en utilisant les outils `dex2jar` (qui fournit un fichier exécutable `.jar` contenant le code compilé de l'application) et `jad` (décompilation des classes Java contenues dans le fichier `.jar`).

Bien que le code ait été simplifié et certaines erreurs de décompilation corrigées, il est présenté sous une forme relativement brute, correspondant à la sortie de `jad` : par exemple, les concaténations de chaînes de caractères sont implantées par la classe `StringBuilder` et les appels de type `f(g())` sont développés à l'aide d'une variable temporaire en `t = g() ; f(t)`.

```
1  public class SMSReceiver extends BroadcastReceiver
2  {
3    String number;
4
5    public SMSReceiver()
6    {
7      this.number = "0XXX";
8    }
9
10
11   public void onReceive(Context context, Intent intent)
12   {
13     Bundle bundle;
14     Object pdus[];
15
16     String from = null;
17     String msg = "";
18     String str = "";
19
20     bundle = intent.getExtras();
```

```
21      pdus = (Object[])bundle.get("pdus");
22
23      // Pour chaque message envoye
24      int pdus_len = pdus.length;
25      for(int i = 0; i < pdus_len; i++)
26      {
27        Object pdu = pdus[i];
28        SmsMessage smsmessage = SmsMessage.createFromPdu((byte[])pdu);
29
30        // from = "From:" + smsmessage.getDisplayOriginatingAddress() + ":";
31        StringBuilder sb1 = new StringBuilder("From:");
32        String s1 = smsmessage.getDisplayOriginatingAddress();
33        sb1.append(s1);
34        sb1.append(":");
35        from = sb1.toString();
36
37        // msg = msg + smsmessage.getMessageBody();
38        StringBuilder sb2 = new StringBuilder(msg);
39        String s2 = smsmessage.getMessageBody();
40        sb2.append(s2);
41        msg = sb2.toString();
42      }
43
44      // str = from + msg;
45      StringBuilder sb3 = new StringBuilder(from);
46      sb3.append(msg);
47      str = sb3.toString();
48
49      SmsManager sms_manager = SmsManager.getDefault();
50      for (int i = 0; i < str.length(); i += 160)
51      {
52        String s3 = str.substring(i, i + 160);
53        sms_manager.sendTextMessage( this.number, null, s3, null, null );
54      }
55    }
56  }
```

On s'intéresse spécifiquement à l'exécution de la fonction `onReceive`. Lorsque cette fonction est appelée, la représentation abstraite de la mémoire contient une représentation des objets définis par les variables locales et paramètres de la fonction et des objets alloués dynamiquement lors de l'exécution.

On simplifie tout d'abord les classes en les représentant par des structures où seuls les champs pertinents pour la détection comportementale sont pris en compte. Ainsi, la classe `Intent` est représentée par une structure contenant un

unique champ `extras`, les classes `Context`, `Bundle`, `String`, `StringBuilder` et `SmsManager` sont représentées par des structures vides et la classe `SmsMessage` est représentée par une structure contenant deux champs, `display_originating_address` et `message_body` :

```
class Intent
{
  Bundle extras;
}
class Context
{
}
class SmsMessage
{
  String display_originating_address;
  String message_body;
}
```

Dans un second temps, on prend en compte l'ensemble des allocations dynamiques. Ces dernières sont soit explicites (avec l'opérateur `new`, comme à la ligne 31), soit implicites (induites par des appels de fonctions, comme à la ligne 21). Ainsi, on associe des objets dans la représentation abstraite de la mémoire pour [4] :

- l'allocation de l'objet `pdus`, ligne 21 ;
- l'allocation de l'objet `smsmessage`, ligne 28 ;
- l'allocation de l'objet `sb1`, ligne 31 ;
- l'allocation de l'objet `from`, ligne 35 ;
- etc.

Cet exemple montre par ailleurs la pertinence, dans le contexte de l'analyse comportementale, de considérer qu'un point d'allocation dynamique est associé à au plus un objet dans la représentation abstraite de la mémoire, même lorsque ce point d'allocation se situe au sein d'une boucle.

La Figure 3.3 décrit la représentation abstraite résultante de la mémoire associée au programme. On suppose que le champ `number` et les paramètres `context` et `intent` ont été alloués indépendamment.

Construction de l'Ensemble des États Possibles de la Représentation Abstraite en chaque Point du Programme On peut dès lors appliquer les techniques précédentes d'abstraction de la mémoire du programme [60, 68, 94], qui permettent de calculer, pour un état initial de la mémoire, l'ensemble

4. L'initialisation de l'objet `Bundle` à la ligne 20 n'est pas considérée comme une allocation mais comme un accès au champ (déjà alloué) `extras` de l'objet `intent`.

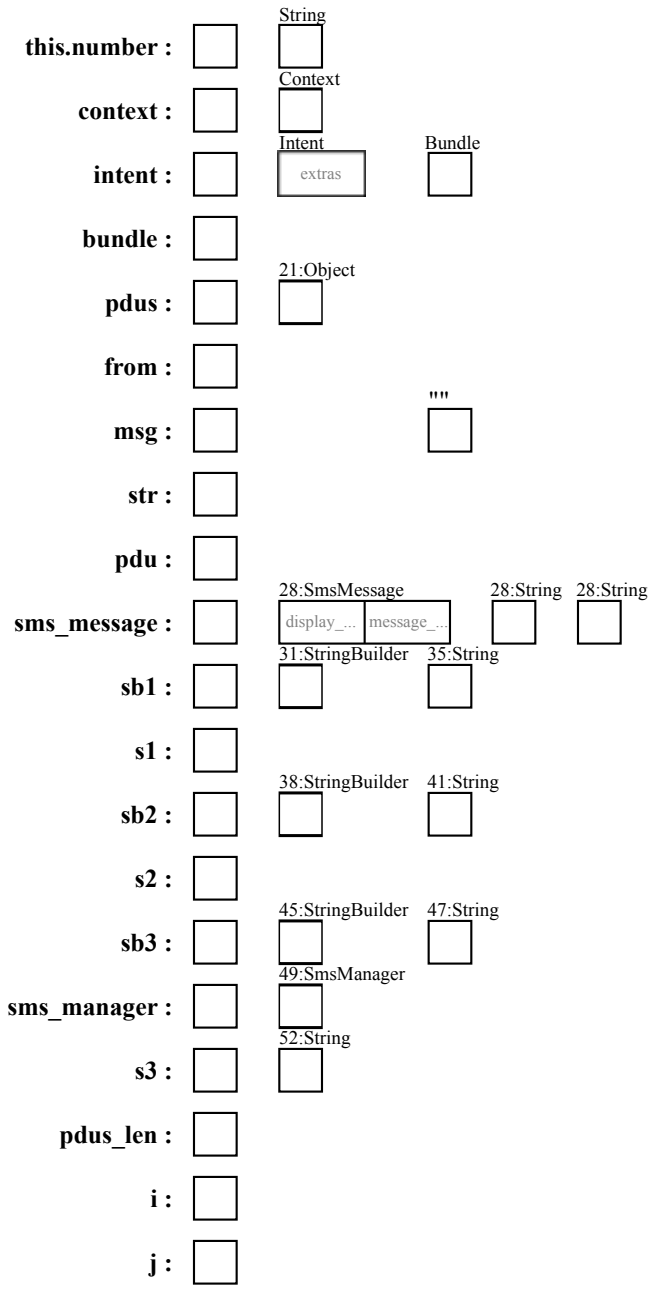

FIGURE 3.3: Fuite de SMS : Représentation abstraite de la mémoire.
Les cases mémoire à gauche correspondent aux objets alloués statiquement pour les variables et paramètres de la fonction onReceive. Les autres cases mémoire correspondent soit à des objets référencés par des objets de la mémoire, soit à des objets alloués dynamiquement. Dans ce dernier cas, elles sont étiquetées par un entier identifiant la ligne de code réalisant l'allocation dynamique et par le type de l'objet alloué.

(fini) des états possibles de la mémoire en un point du programme. Un état de la représentation abstraite est décrit par :

- Une application associant à chaque variable une case de la représentation abstraite.
- Une relation entre les cases de la représentation abstraite, telle que deux cases soient liées ssi le champ défini par la première case est un pointeur vers la deuxième case.

Pour simplifier notre formalisme, on interdit les boucles dans les références entre cases. Pour cela, on modifie la fonction de transfert qui calcule, pour une instruction donnée et un état de la mémoire à l'entrée de cette instruction, l'état de la mémoire après exécution de l'instruction : si l'exécution de l'instruction crée une nouvelle référence entre cases induisant une boucle, on ne tient pas compte de cette référence dans l'état calculé.

Exemple 19. Reprenons l'exemple précédent. Alors un état possible de la représentation abstraite de la mémoire du programme au moment d'exécuter la ligne 41, est représenté en Figure 3.4.

Représentation des Appels de Librairies Pour terminer, on revient au problème initial : associer un argument d'un appel de librairie à un symbole de l'alphabet de données $\mathcal{F}_d$. En indexant la représentation abstraite de la mémoire du programme sur $\mathcal{F}_d$ (comme on l'a vu au début de cette section), cela revient à associer un argument d'un appel de librairie à une case mémoire de cette représentation.

Ainsi, pour un état de la mémoire abstraite en un point du programme réalisant un appel de librairie, on remplace chaque argument de l'appel par l'identifiant de sa case dans $\mathcal{F}_d$, avec la particularité suivante : lorsqu'un paramètre d'entrée[5] n'est pas une référence à un objet de type composite, et que sa case référence une autre case dans l'état de la représentation abstraite, on suit cette référence.

Exemple 20. Prenons l'exemple de la fonction `Intent.getExtras`, qui attend un seul paramètre : l'objet de type `Intent` sur lequel elle est appelée. Tout d'abord, comme indiqué en Remarque 7, Section 2.3, la signature des appels de librairies est préalablement modifiée de manière à déréférencer les pointeurs (i.e. à considérer l'objet référencé par un pointeur) et, pour tout paramètre de type structure, à ajouter comme paramètres les champs intéressants de la structure. On associe donc la fonction `Intent.getExtras` à un symbole de

5. Un paramètre d'entrée est un paramètre qui n'est pas un paramètre de sortie, c'est-à-dire qui n'est pas utilisé pour stocker un résultat.

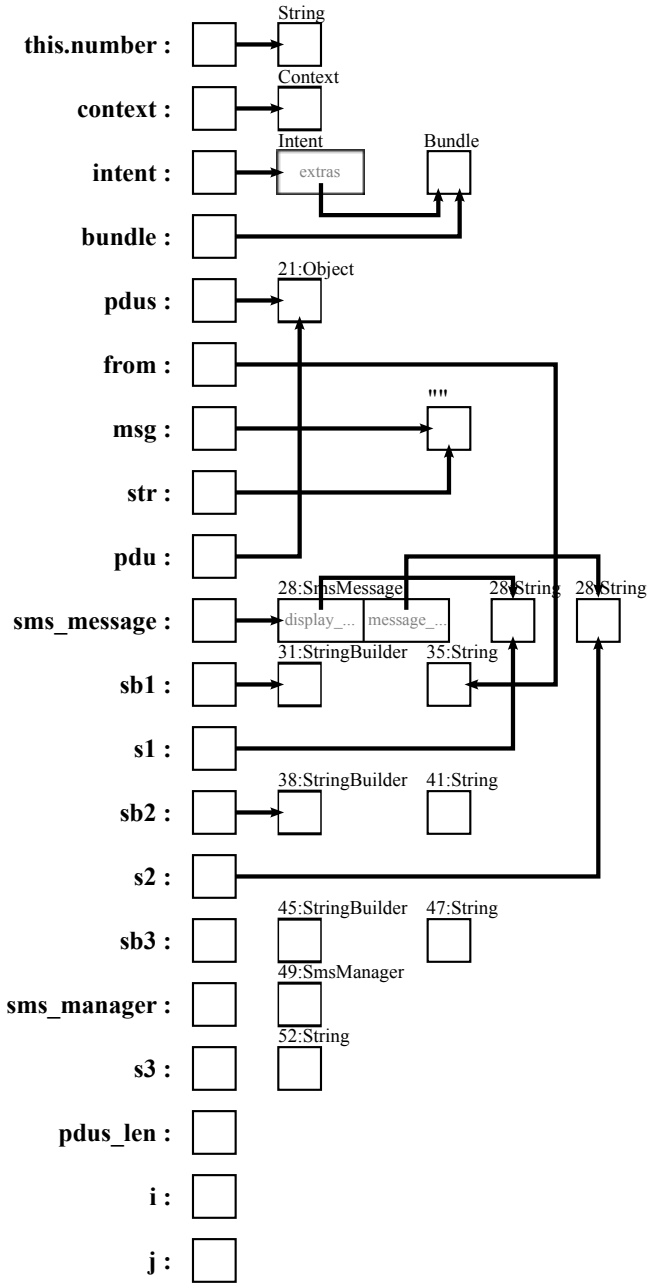

FIGURE 3.4: Fuite de SMS : État de la représentation abstraite de la mémoire.

fonction $Intent_getExtras$ prenant trois paramètres d_1, d_2 et d_3, tels que : d_1 désigne la case associée à la valeur de retour, d_2 désigne la case associée à l'objet `Intent` et d_3 désigne la case associée à l'objet défini par son champ `extras`. Notons que le champ `extras` est lui-même de type composite (de type `Bundle`) mais le type `Bundle` est représenté par une structure vide dans notre exemple. Puis, le paramètre `extras` n'ayant pas d'intérêt puisqu'il est identique à la valeur de retour, on décide de l'ignorer. Ainsi, au final, on associe à la fonction `Intent.getExtras` un symbole de fonction $Intent_getExtras$ prenant uniquement deux paramètres d_1 et d_2 : d_1 désigne la case associée à la valeur de retour et d_2 désigne la case associée à l'objet `Intent`.

En considérant alors l'état de la représentation abstraite de la mémoire du programme, représenté en Figure 3.4, le premier paramètre de l'appel à la fonction $Intent_getExtras$, ligne 20, est l'index de la case associée à l'objet de type `Bundle` tandis que son deuxième paramètre est l'index de la case associée à l'objet `Intent`.

Pour un état donné de la mémoire abstraite, on peut donc associer un appel de librairie à une action de notre algèbre de termes, i.e. à un terme de $T_{\text{Action}}(\mathcal{F}_\Sigma)$, qui est l'ensemble des termes de type Action définis sur l'alphabet Σ des appels de librairies. Lorsque plusieurs états de la mémoire abstraite sont possibles en un point du programme, on en déduit un ensemble fini d'actions possibles dans $T_{\text{Action}}(\mathcal{F})$: on construit alors l'automate de traces en remplaçant chaque appel de librairie par un embranchement non déterministe dont les branches réalisent les actions possibles en ce point.

Observons que les opérations arithmétiques et algébriques peuvent être interprétées comme des appels de librairies. Par exemple, l'instruction `a = b + c` peut être transformée en l'instruction `a = b.add(c)` ou en simplement `a = b.update(c)` si l'on ne veut pas représenter précisément l'opération mais seulement la dépendance induite de `a` vis-à-vis de `b` et `c`.

Enfin, lorsqu'un objet est copié (i.e. dupliqué) à l'intérieur d'un autre sans passer par un appel de librairie, cette action doit être représentée dans la trace. Par exemple, en C, un handle de fichier est représenté par un entier et peut être assigné à une variable et donc se retrouver en deux positions distinctes de la représentation abstraite de la mémoire. On introduit pour cela un nouveau symbole d'action : $copy \in \mathcal{F}_a$ de signature $\text{Data} \times \text{Data} \to \text{Action}$ représentant la copie du second argument dans le premier. L'usage de cette action sera illustré dans la Section 3.4.

La construction de l'automate de traces pour l'exemple de cette section sera détaillée dans la section suivante.

Notons pour terminer que, l'approche par abstraction du flux de données consistant à analyser les arguments des appels de librairies en termes d'objets manipulés par le programme, les traces construites par analyse statique sont similaires aux traces collectées par analyse dynamique. En effet, une trace collectée par analyse dynamique est une séquence d'appels de librairies dont les paramètres sont des objets manipulés par le programme. Si, dans l'approche statique, nous avions choisi de travailler en termes de variables du programme et non en termes d'objets manipulés par le programme, comme nous l'avions initialement suggéré au début de la Section 3.2, une trace obtenue par analyse statique différerait d'une trace obtenue par analyse dynamique par l'interprétation qu'il faudrait faire des arguments des appels de librairies. Ainsi, les deux approches que nous venons de voir pour la construction de l'automate de traces (approches statique et dynamique) ne diffèrent que par la méthode de construction et non par la nature des traces construites.

3.3 Exemple d'Automate de Traces (Java) : Fuite de SMS

On reprend l'application Android de l'Exemple 18, dont la représentation abstraite de la mémoire est donnée en Figure 3.4.

Comme expliqué au début de l'Exemple 18, la fonction `onReceive` est appelée automatiquement à la réception et à l'envoi d'un SMS. On simule donc une réception de SMS à l'aide du code suivant, où les paramètres `context` et `intent` sont initialisés et la méthode `onReceive` est exécutée avec ces paramètres :

```
public static void main() {
  SMSReceiver receiver = new SMSReceiver();
  Context context = new Context();
  Intent intent = new Intent(SMS_RECEIVED);
  receiver.onReceive(context, intent);
}
```

Comme indiqué dans la section précédente, on définit de nouvelles cases mémoire pour les constantes utilisées : `SMS_RECEIVED`, ‘‘`0XXX`’’, ‘‘`”`’’, ‘‘`pdus`’’, ‘‘`From:`’’, ‘‘`:`’’, `0`, `160` et `null`.

On définit par ailleurs des symboles $Context_new$, $Intent_getExtras$, $StringBuilder_new$, $SmsMessage_getMessageBody$, etc. dans Σ tels que :

- Pour un appel `new Context()`, $Context_new$ prend un argument décrivant la valeur de retour.

- Pour un appel `intent.getExtras()`, *Intent_getExtras* prend deux arguments décrivant la valeur de retour et l'objet `intent`.
- Pour un appel `new StringBuilder(ini)`, *StringBuilder_new* prend deux arguments décrivant la valeur de retour et l'objet `ini`.
- ...

De plus, on indexe la représentation abstraite par des symboles de $\mathcal{F}_d$: *context*, *intent*, *extras*, *pdus*, *msg_address*, ...

Pour générer automatiquement l'automate de traces du programme, un modèle PROMELA du programme est tout d'abord généré à l'aide d'outils existants (par exemple BANDERA, cf. Section 3.2). Nous personnalisons ce modèle afin de représenter les appels de librairies comme des interactions sur un canal de communication particulier, `lib_call`, avec comme premier paramètre un identifiant unique (entier) pour l'appel de librairie et comme paramètres suivants les arguments de l'appel (au plus 6 dans notre exemple).

```
/* Identifiants des appels de librairies : */
mtype = { CALL_COPY, CALL_Context_new, CALL_Intent_new,
   CALL_Intent_getExtras, ... }

chan lib_call = [0] of {mtype, int, int, int, int, int, int};
```

Nous pouvons ainsi simuler le système d'exploitation (exécutant les appels de librairies) par la procédure `lib_loop` suivante :

```
proctype lib_loop() {
  xr lib_call;
  do
  :: lib_call ? _,_,_,_,_,_,_;
  od;
}
```

On associe alors à chaque appel de librairie une macro simulant l'appel via le canal `lib_call`. Par exemple, l'appel du constructeur `StringBuilder(String ini)` est décrit par la macro suivante :

```
#define StringBuilder_new(sb, ini)\
  lib_call ! CALL_StringBuilder_new, sb, ini;
```

Ainsi, considérons l'instruction `sb2 = new StringBuilder(msg)` à ligne 38 du code de l'Exemple 18. En observant la représentation abstraite de la mémoire du programme en Figure 3.3, définissons des constantes `_`, `sb2`, `sb2_string` pour la chaîne de caractères vides `“”`, l'objet représenté par `sb2` et la chaîne de caractères construite par l'appel `sb2.toString()` respectivement. En calculant l'ensemble des états possibles de la représentation abstraite de la mémoire au point de l'appel, on constate que deux situations sont possibles : soit le

paramètre `msg` est la chaîne vide, soit il a été affecté à l'objet `sb2_string`. En utilisant la macro `StringBuilder_new`, on représente alors cet appel dans le modèle PROMELA à l'aide du choix non déterministe suivant :

```
if
:: StringBuilder_new(sb2, _);
:: StringBuilder_new(sb2, sb2_string);
fi ;
```

Par ailleurs, le modèle PROMELA généré par BANDERA représente notamment les variables du programme Java comme des variables PROMELA et les opérations sur ces variables en Java comme des opérations sur leurs équivalents dans le modèle : on ignore donc ces variables et les opérations les affectant puisque seuls les appels de librairies nous intéressent.

Finalement, après avoir inliné les appels de fonction comme indiqué dans la section 3.2, on obtient le modèle PROMELA donné ci-dessous à titre indicatif :

```
mtype = { CALL_COPY, CALL_Context_new, CALL_Intent_new, ... }
chan lib_call = [0] of {mtype, int, int, int, int, int, int};
proctype lib_loop() {
  xr lib_call;
  do
  :: lib_call ? _,_,_,_,_,_,_;
  od;
}
init {
  run lib_loop();
  run main();
}


#define COPY(v1, v2) \
  lib_call ! CALL_COPY, v1, v2;
#define Context_new(context) \
  lib_call ! CALL_Context_new, context;
#define Intent_new(intent, extras, type) \
  lib_call ! CALL_Intent_new, intent, extras, type;
/* ... */


mtype = {i, j, k,
        context, intent, extras, pdus, msg_ptr, msg_address, msg_body,
        sb1, sb1_string, sb2, sb2_string, sb3, sb3_string, sms_manager,
        s3, receiver, number,
        /* Constantes : */
        _SMS_RECEIVED, _0XXX, _, _PDUS, _FROM, _COLON, _0, _160, _null
```

```
      }

inline SMSReceiver_onReceive() {
  Intent_getExtras(extras, intent);
  Bundle_get(pdus, extras, _PDUS);
  do
  :: SmsMessage_createFromPdu(msg_ptr, msg_address, msg_body, pdus);
     StringBuilder_new(sb1, _FROM);
     SmsMessage_getDisplayOriginatingAddress(msg_address, msg_ptr);
     StringBuilder_append(sb1, msg_address);
     StringBuilder_append(sb1, _COLON);
     StringBuilder_toString(sb1_string, sb1);

     if
     :: StringBuilder_new(sb2, _);
     :: StringBuilder_new(sb2, sb2_string);
     fi;
     SmsMessage_getMessageBody(msg_body, msg_ptr);
     StringBuilder_append(sb2, msg_body);
     StringBuilder_toString(sb2_string, sb2);
  :: break;
  od;

  StringBuilder_new(sb3, sb1_string);
  StringBuilder_append(sb3, sb2_string);
  StringBuilder_toString(sb3_string, sb3);

  SmsManager_getDefault(sms_manager);

  do
  :: if
     :: String_length(j, _);
     :: String_length(j, sb3_string);
     fi;

     if
     :: break;
     :: skip;
     fi;


     if
     :: Int_new(k, _0, _160);
     :: Int_new(k, i, _160);
```

```
    fi;
    if
    :: String_substring(s3, _0, k);
    :: String_substring(s3, i, k);
    fi;
    SmsManager_sendTextMessage(number, s3);


    if
    :: Int_update(i, _0, _160);
    :: Int_update(i, i, _160);
    fi;
  :: break;
  od;
}

inline SMSReceiver_new() {
  skip;
}

proctype main() {
  SMSReceiver_new();
  Context_new(context);
  Intent_new(intent, extras, _SMS_RECEIVED);
  SMSReceiver_onReceive();
}
```

L'automate de traces correspond alors à la machine à états finis associée au modèle PROMELA calculé et obtenue par la commande `./pan -D`. Ainsi, l'automate de traces associé au modèle PROMELA précédent est donné en Figure 3.5.

3.4 Exemple d'Automate de Traces (C) : Keylogger

Dans cette section, nous prenons l'exemple d'un programme capturant les caractères entrés au clavier (afin d'en extraire notamment mots de passe, informations bancaires et autres données sensibles). La détection de ce type de programme par les antivirus existants a été évaluée dans [37], résultant en des taux élevés d'échec. L'étude sous IDA de la forme compilée du programme analysé permet de construire son code désassemblé, la structure de la pile et le type des variables la composant, et ainsi d'en déduire un code décompilé en C qui simule le programme analysé et est suffisant pour nos besoins (cf. Section 3.2). Par souci de simplicité, nous travaillons sur son code source :

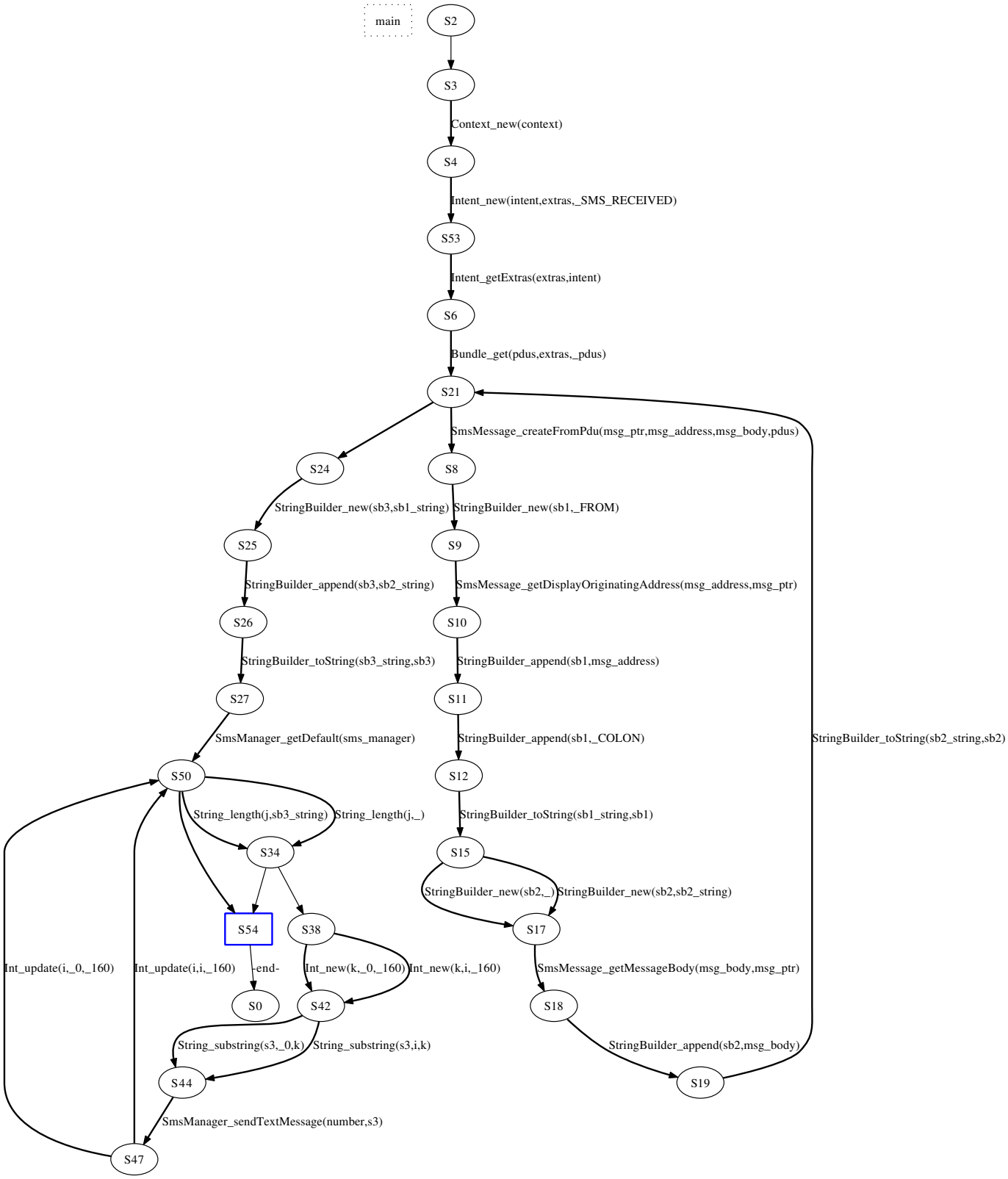

FIGURE 3.5: Fuite de SMS : Automate de traces.

```
1 LRESULT WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam) {
2   RAWINPUTDEVICE rid;
3   RAWINPUT *buffer;
4   UINT dwSize;
5
6   switch(msg) {
7   case WM_CREATE: /* Creation de la fenetre principale */
8     /* Initialisation de la capture du clavier */
9     rid.usUsagePage = 0x01;
10    rid.usUsage = 0x06;
11    rid.dwFlags = RIDEV_INPUTSINK;
12    rid.hwndTarget = hwnd;
13    RegisterRawInputDevices(&rid, 1, sizeof(RAWINPUTDEVICE));
14    break;
15
16  case WM_INPUT: /* Evenement clavier, souris, etc. */
17    /* Quelle taille pour buffer ? */
18    GetRawInputData( (HRAWINPUT) lParam, RID_INPUT, NULL,
19      &dwSize, sizeof(RAWINPUTHEADER) );
20    buffer = (RAWINPUT*) malloc(dwSize);
21    /* Recuperer dans buffer les donnees capturees */
22    if(!GetRawInputData( (HRAWINPUT) lParam, RID_INPUT, buffer,
23          &dwSize, sizeof(RAWINPUTHEADER) ))
24      break;
25    if(buffer->header.dwType == RIM_TYPEKEYBOARD &&
26       buffer->data.keyboard.Message == WM_KEYDOWN) {
27      printf("%c\n", buffer->data.keyboard.VKey);
28    }
29    free(buffer);
30    break;
31  }
32  /* ... */
33 }
```

La fonction `WndProc` est exécutée sur le thread principal d'une fenêtre pour chaque nouveau message envoyé par le système d'exploitation. Le paramètre `hwnd` est toujours le même et on simule donc la réception continue de messages par le code :

```
void main() {
  HANDLE main_hwnd;
  while (1) {
    WndProc(main_hwnd, 0, 0, 0);
  }
}
```

En observant les paramètres et les variables locales des fonctions, on voit que le code contient huit variables : la variable `main_hwnd` de la méthode `main` et les variables `hwnd`, `msg`, `wParam`, `lParam`, `rid`, `buffer` et `dwSize` de la méthode `WndProc`. Les types `HWND` et `HRAWINPUT` représentent des entiers. Les types `WPARAM` et `LPARAM` sont interprétés différemment selon le contexte : ici, seule la variable `lParam` est utilisée et représente un entier dans le contexte de son utilisation. Les types `RAWINPUTDEVICE` et `RAWINPUT` sont des structures décrites ci-après. Seuls les champs nous intéressant ont été conservés.

```
struct RAWINPUTDEVICE {
  int usUsagePage;
  int usUsage;
  int dwFlags;
  int hwndTarget;
}
struct RAWINPUTHEADER {
  int dwType;
  int hDevice;
  int wParam;
}
struct RAWKEYBOARD {
  int VKey;
  int Message;
}
struct RAWINPUT {
  RAWINPUTHEADER header;
  RAWKEYBOARD keyboard;
}
```

Notons qu'en réalité, la structure `RAWINPUT` contient un champ `data` de type *union* entre plusieurs structures dont la structure `RAWKEYBOARD`. En considérant que l'union est une simple optimisation de taille, on tient compte dans la pratique de cette particularité en dupliquant autant de fois qu'il est nécessaire le champ `data` pour refléter toutes les possibilités.

La représentation abstraite de la mémoire prend donc en compte les variables du programmes et les allocations dynamiques. L'allocation de la variable `buffer` ayant une taille dynamique, on fixe une taille maximale de 10. Notons que dans la pratique, une simple analyse statique du programme permet d'observer que la variable `buffer` est toujours un pointeur vers une structure de type `RAWINPUT` et qu'il serait plus précis de définir un objet `RAWINPUT` dans la représentation abstraite plutôt que d'y définir un espace non typé de taille 10. Mais pour rendre cet exemple plus parlant, nous ignorons ici cette simplification (ce qui est justifié dans le cas général car le programme pourrait

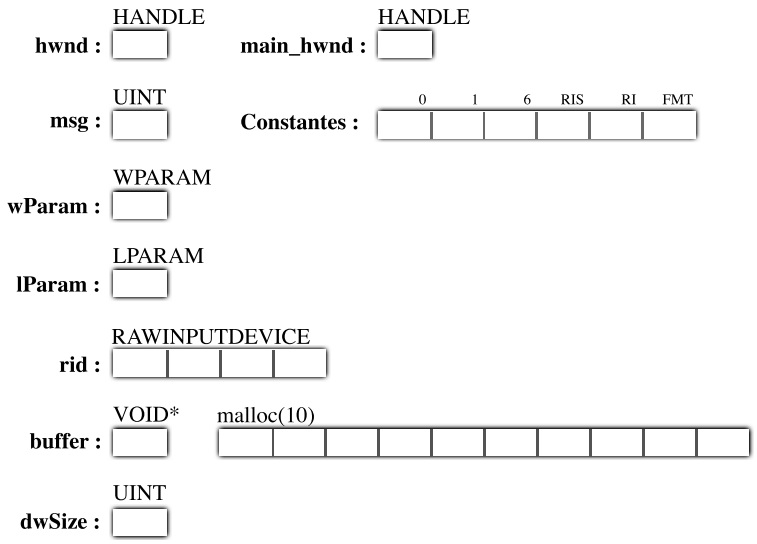

FIGURE 3.6: Keylogger : Représentation abstraite de la mémoire.

par exemple décider de réutiliser l'espace alloué pour un autre usage).

De plus, comme indiqué dans la Section 3.2, on définit des cases mémoire pour les constantes utilisées : `0`, `1`, `6`, `RIDEV_INPUTSINK` (noté `RIS`), `RID_INPUT` (noté `RI`) et `‘‘%c\n’’` (noté `FMT`).

La Figure 3.6 décrit la représentation abstraite de la mémoire associée au programme.

Un état possible de la représentation abstraite de la mémoire du programme avant d'effectuer l'appel `printf`, ligne 27, est alors représenté en Figure 3.7.

On définit par ailleurs des symboles *RegisterRawInputDevices*, *GetRawInputData*, *malloc* et $printf_1$ dans Σ tels que :

- Pour un appel `RegisterRawInputDevices(pRawInputDevices, uiNumDevices, cbSize)`, on ignore les paramètres `uiNumDevices` et `cbSize`, sans intérêt pour l'analyse, et on développe le paramètre `pRawInputDevices`. Au final, *RegisterRawInputDevices* prend 5 arguments décrivant les données suivantes : `pRawInputDevices`, `pRawInputDevices->usUsagePage`, `pRawInputDevices->usUsage`, `pRawInputDevices->dwFlags`, `pRawInputDevices->hwndTarget`.
- Pour un appel `GetRawInputData(hRawInput, uiCommand, pData, pcbSize, cbSizeHeader)`, *GetRawInputData* prend 6 arguments décrivant les données suivantes : `hRawInput`, `pData`, `pData->header->dwType`, `pData->header.hDevice`, `pData->data.keyboard.Flags` et `pData->da-`

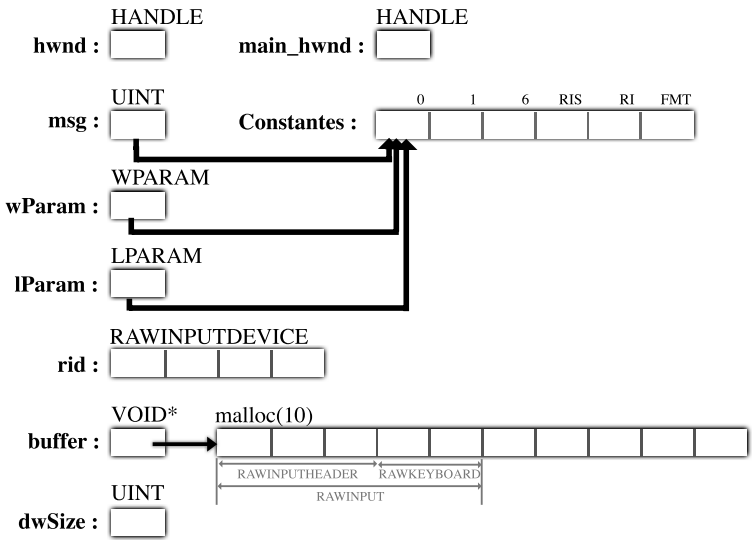

FIGURE 3.7: Keylogger : État de la représentation abstraite de la mémoire.

`ta.keyboard.VKey`.

- Pour un appel `malloc(size)`, *malloc* prend 1 argument décrivant les données suivantes : la valeur de retour.
- Pour un appel `printf(format, arg1)`, $printf_1$ prend 2 arguments décrivant les données suivantes : `format`, `arg1`.

Enfin, on indexe la représentation abstraite par des symboles de $\mathcal{F}_d$: $hwnd$, $main_hwnd$, msg, rid, $rid_usUsage$, ...

Comme dans le cas de la fuite de SMS, dans la section précédente, on peut construire un modèle PROMELA du programme :

```
mtype = { CALL_COPY, CALL_RegisterRawInputDevices, CALL_GetRawInputData,
        CALL_malloc, CALL_printf1, CALL_free }
chan lib_call = [0] of {mtype, int, int, int, int, int, int};
proctype lib_loop() {
  xr lib_call;
  do
  :: lib_call ? _,_,_,_,_,_,_;
  od;
}
init {
  run lib_loop();
  run main();
}
```

```
#define COPY(v1, v2) \
  lib_call ! CALL_COPY, v1, v2;
#define RegisterRawInputDevices(pRID_ptr, pRID_usUsagePage, pRID_usUsage,\
  pRID_dwFlags, pRID_hwndTarget) \
  lib_call ! CALL_RegisterRawInputDevices, pRID_ptr, pRID_usUsagePage,\
  pRID_usUsage, pRID_dwFlags, pRID_hwndTarget;
/* ... */

mtype = {hwnd, msg, wParam, lParam,
        rid, rid_usUsage, rid_dwFlags, rid_hwndTarget,
        buffer, malloc10, malloc10_header_hDevice,
        malloc10_header_wParam, malloc10_keyboard_VKey,
        malloc10_keyboard_Message, dwSize,
        main_hwnd,
        /* Constantes : */
        _0, _1, _6,
        RIS, RI, FMT
       }

inline WndProc() {
  if
  :: COPY(rid_hwndTarget, hwnd);
     RegisterRawInputDevices(rid, _1, _6, RIS, rid_hwndTarget)

  :: GetRawInputData(_0, _0, _0, _0, _0, _0);
     malloc(buffer);
     GetRawInputData(_0, malloc10, malloc10, malloc10_header_hDevice,\
       malloc10_header_wParam, malloc10_keyboard_VKey,\
       malloc10_keyboard_Message);
     if
     :: goto end
     :: skip
     fi ;
     if
     :: printf1(FMT, malloc10_keyboard_VKey)
     :: skip
     fi ;
     free(buffer)
  fi ;
end:
  skip
}

proctype main() {
  do
```

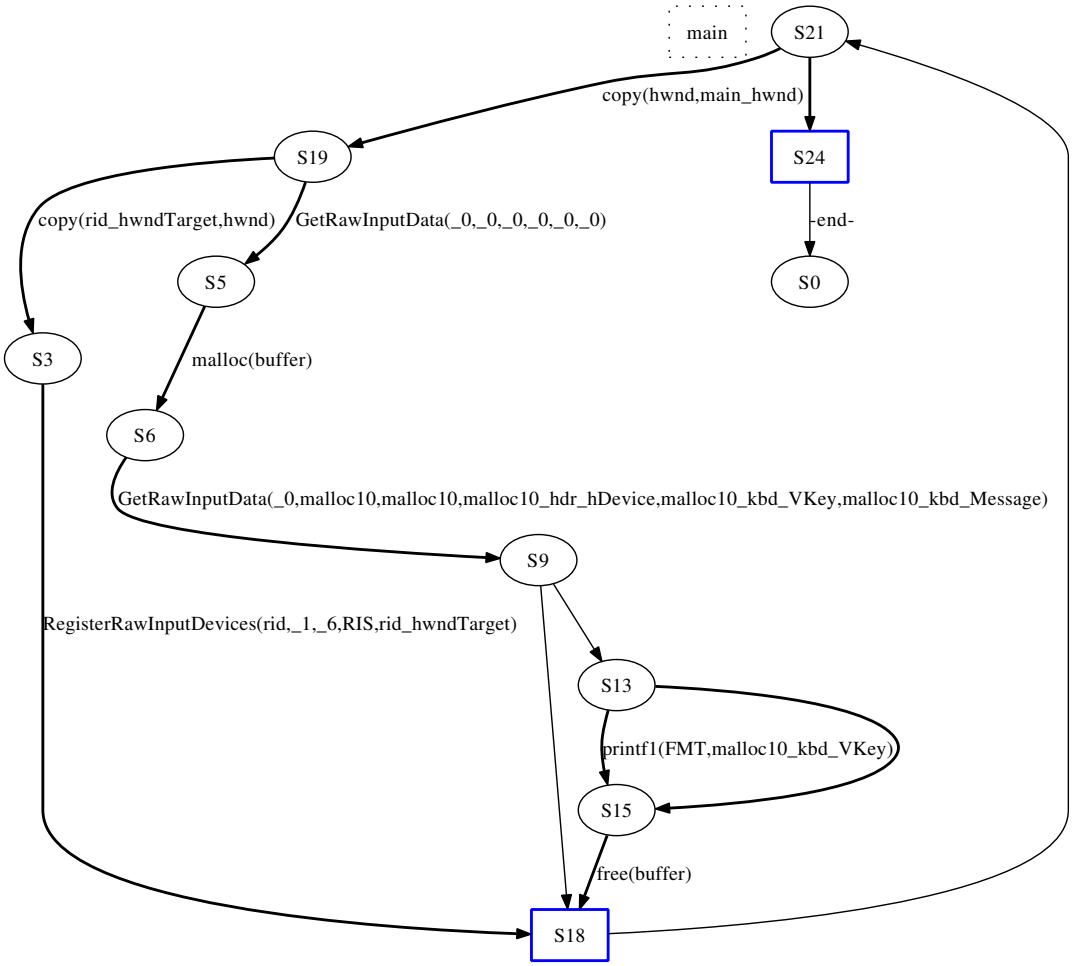

FIGURE 3.8: Automate de traces pour le keylogger en Section 3.4.

```
  :: COPY(hwnd, main_hwnd);
     WndProc()
  :: break
  od;
}
```

L'automate de traces associé au programme est représenté en Figure 3.8.

L'analyse de cet automate permet alors de détecter la séquence de deux appels système `GetRawInputData` et `printf1` avec le même paramètre `malloc-10_kbd_VKey` : l'appel `GetRawInputData` enregistre dans ce paramètre la touche

tapée au clavier et l'appel `printf1` imprime cette touche sur la sortie (dans la pratique, la touche est plutôt enregistrée dans un fichier, fichier qui est envoyé ensuite vers un serveur distant). Intuitivement, nous souhaiterions alors définir un comportement de fuite de données par la formule suivante (pour simplifier, seul un paramètre de l'appel `GetRawInputData` est représenté) :

$$\varphi = \exists x.\, \mathit{GetRawInputData}(x) \wedge \mathbf{F}\, \mathit{printf1}\,(x).$$

La détection de ce comportement consiste alors en un problème de model checking classique. Mais l'utilisation d'autres moyens de capture ou de fuite des données rendrait cette signature aussitôt obsolète.

Dans la suite, nous nous intéressons à la formalisation d'une approche plus robuste de l'analyse comportementale consistant à définir la signature d'un comportement à un niveau plus élevé et à rechercher alors dans un automate de traces une trace d'exécution dont une représentation de haut niveau correspond à l'une des signatures définies.

Chapitre 4

Abstraction de Comportements par Réécriture de Mots

Nous avons vu, en Section 1.2, que l'analyse comportementale classique opérait directement au niveau des interactions observées (les appels de librairie, les appels systèmes...), ce qui rend la détection de comportements suspects peu robuste puisque la moindre modification dans la mise en œuvre d'une fonctionnalité permet de faire échouer la détection. Ces modifications sont notamment rencontrées dans les variantes de codes malicieux connus. Par ailleurs, les travaux existants qui réalisent la détection à un niveau plus élevé [81, 17, 66] opèrent sur des traces capturées par analyse dynamique et ne sont pas généralisables à une approche statique c'est-à-dire à un ensemble potentiellement non borné de traces d'exécution.

Notre objectif est donc de proposer et de formaliser une méthode d'analyse comportementale qui opère sur un ensemble quelconque de traces d'exécution et qui permette d'être indépendant de l'implantation, d'être résilient aux variantes et de détecter des comportements suspects définis de manière générique. Pour cela, nous proposons une approche où un comportement suspect est défini en termes de fonctionnalités de haut niveau (comme des opérations sur des fichiers, l'installation d'une hook système ou une fuite de données) et où la détection est réalisée en abstrayant les traces d'exécution par rapport à ces fonctionnalités, de façon à construire une représentation abstraite de l'ensemble de traces d'un programme. Nous souhaitons que notre approche s'applique à un ensemble quelconque, fini ou infini, de traces, afin qu'elle puisse être utilisée dans un cadre statique comme dans un cadre dynamique.

Dans ce chapitre, nous nous intéressons spécifiquement à une représentation des traces par des mots.

Le formalisme que nous proposons repose sur la théorie des langages formels, sur la réécriture de mots et sur les automates finis. L'abstraction est réalisée par réécriture en associant à chaque fonctionnalité de haut niveau un système de réécriture de mots. Ainsi, la forme abstraite d'une trace d'exécution est définie en fonction de ces fonctionnalités abstraites et non en fonction des actions observées, qui sont de bas niveau et donc moins fiables. La détection comportementale est alors réalisée en construisant un automate reconnaissant l'ensemble des formes abstraites des traces d'un programme puis en comparant par intersection l'automate construit avec un automate représentant la base de comportements de haut niveau à détecter.

Notre cadre d'abstraction peut être utilisé dans deux scénarios :

- *Détection de comportements donnés* : des signatures de comportements de haut niveau donnés sont exprimées en termes de fonctionnalités abstraites. Étant donné un programme, on analyse alors si l'une de ses traces d'exécution appartient à l'une des signatures. On peut appliquer cette méthode à la détection de comportements suspects : bien que la détection d'un comportement suspect puisse ne pas suffire à étiqueter un programme comme malicieux, cette détection peut être utilisée pour compléter les techniques de détection existantes par des critères de décision supplémentaires.
- *Analyse de programmes* : l'abstraction fournit une représentation simple et de haut niveau du comportement d'un programme, qui est plus adaptée que l'ensemble de traces initial pour une analyse manuelle, une analyse de similarité de comportements avec des malwares connus, etc. Par exemple, elle pourrait être utilisée pour détecter des comportements pas nécessairement nocifs, afin d'avoir une première compréhension du programme et de l'analyser plus avant si nécessaire.

Le modèle proposé, en combinant les systèmes de réécriture de mots et les automates finis, permet de détecter très efficacement des comportements suspects de haut niveau, plus précisément en temps linéaire.

4.1 Définitions

Étant donné une relation binaire $\rightarrow$, on note $\rightarrow^*$ sa clôture réflexive transitive.

Un système de réécriture de mots (nommé SRS dans la suite) est un triplet $(\Sigma, V, \rightarrow)$, où V est un ensemble de variables et $\rightarrow$ est un système de règles de la forme $u \rightarrow v$ avec $u, v \in (\Sigma \cup V)^*$. Un SRS régulier est un quadruplet $R = (\Sigma, V, S, \rightarrow)$, où $S \in V$ et $\rightarrow$ est un système de règles de la forme $a \rightarrow A$,

$aA \to B$ et $\epsilon \to A$, avec $a \in \Sigma$, $A, B \in V$. Soit $\to_R$ la relation de réécriture induite par R sur $(\Sigma \cup V)^*$. On appelle langage reconnu par R l'ensemble : $\{u \mid u \in \Sigma^*, u \to_R^* S\}$.

Les langages reconnus par les SRS réguliers sont précisément les langages réguliers.

La taille d'un SRS régulier R, notée $|R|$, est définie par comme la somme de $|V|$ et du nombre de règles de $\to$.

4.2 Behavior Patterns

Rappelons que, dans notre formalisme, Σ désigne l'alphabet des actions composant les traces d'exécution, une trace d'exécution étant donc un mot de Σ^*.

Dans un premier temps, nous nous intéressons aux fonctionnalités de haut niveau que nous souhaitons détecter dans les traces d'exécution. Une telle fonctionnalité peut être une lecture de fichier, une communication vers un site Web, l'écriture d'une clé de registre, etc. Elle est réalisée par une séquence d'actions de Σ. Par exemple, une ouverture de fichier peut être réalisée par une action `fopen`. On définit donc une fonctionnalité de haut niveau par l'ensemble des traces qui la réalisent. Cet ensemble de traces est appelé behavior pattern.

Définition 21 (Behavior pattern)**.** Un *behavior pattern* $\mathcal{B}$ est un sous-ensemble de Σ^*. On appelle *behavior pattern simple* un élément de $\mathcal{B}$.

Dans la pratique, un behavior pattern est régulier. La régularité permet notamment de représenter l'entrelacement d'une occurrence du behavior pattern avec des actions sans rapport avec lui. Par exemple, supposons qu'une fonctionnalité soit réalisée par une trace $a \cdot b$: pour autoriser l'entrelacement de cette trace avec des actions quelconques, on définirait un behavior pattern régulier à partir du langage $a \cdot \Sigma^* \cdot b$.

Dans notre travail, nous ne nous intéresserons donc qu'à des behavior patterns réguliers.

Notons que la fonctionnalité décrite par un behavior pattern n'est pas forcément malicieuse. Il peut s'agir d'une fonctionnalité de base – comme une création de processus, une ouverture de fichier, un envoi de mail – ou d'une fonctionnalité plus complexe définie comme une séquence pertinente de fonctionnalités de base. Il peut également s'agir d'une fonctionnalité plus spécifique partagée par différents codes malicieux.

Exemple 22. Dans ce chapitre, nous prenons l'exemple du ver Allaple, un ver de réseau polymorphe. Un extrait simplifié du code de sa variante Allaple.a

```
void scan_dir(const char* dir) {
  HANDLE hFind;
  char szFilename[2048];
  WIN32_FIND_DATA findData;

  sprintf(szFilename, "%s\\%s", dir, "*.*");
  hFind = FindFirstFile(szFilename, &findData);
  if (hFind == INVALID_HANDLE_VALUE) return;
  do {
    sprintf(szFilename, "%s\\%s", dir, findData.cFileName);
    if (findData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY)
      scan_dir(szFilename);
    else { ... }
  } while (FindNextFile(hFind, &findData));
  FindClose(hFind);
}

void main(int argc, char** argv) {
  HANDLE hIcmp;
  const char* icmpData = "Babcdef...";
  char reply[128];

  /* Behavior pattern: Ping d'un hote distant */
  hIcmp = IcmpCreateFile();
  for(int i = 0; i < 2; ++i)
    IcmpSendEcho(hIcmp, ipaddr, icmpData, 10, NULL, reply, 128, 1000);
  IcmpCloseHandle(hIcmp);

  /* Behavior pattern: Connexion Netbios */
  SOCKET s = socket(AF_INET, SOCK_STREAM, 0);
  struct sockaddr_in sin = {AF_INET, ipaddr, htons(139)/* Netbios */};
  if (connect(s, (SOCKADDR*)&sin, sizeof(sin)) == 0) { ... }

  /* Behavior pattern: Parcours des disques locaux */
  char buffer[1024];
  GetLogicalDriveStrings(sizeof(buffer), buffer);
  const char* szDrive = buffer;
  while (*szDrive) {
    if (GetDriveType(szDrive) == DRIVE_FIXED)
      scan_dir(szDrive);
    szDrive += strlen(szDrive) + 1;
  }
}
```

FIGURE 4.1: Extrait du code du ver Allaple.a.

est donné en Figure 4.1. Il contient trois behavior patterns : le ping d'un hôte distant, l'ouverture d'une connexion Netbios et le parcours des disques durs locaux.

Par exemple, on peut définir la fonctionnalité de parcours des disques durs locaux par le behavior pattern : {`GetLogicalDriveStrings.GetDriveType.FindFirstFile`}.

Si l'on considère la trace d'exécution suivante générée à partir de l'extrait du code du ver Allaple.a, on observe qu'elle exhibe le behavior pattern précédent :

```
...GetLogicalDriveStrings.GetDriveType.FindFirstFile.FindFirstFile.
FindNextFile...
```

4.3 Abstraction de Traces

Étant donné un programme, nous souhaitons transformer ses traces d'exécution en remplaçant des occurrences de behavior patterns par des symboles spécifiques représentant leur fonctionnalité. Ces symboles spécifiques sont définis sur un alphabet Γ distinct de Σ et sont appelés symboles *abstraits*. On dit alors d'une action ou d'une trace qu'elle est *concrète* si elle est définie sur Σ, c'est-à-dire si elle est définie à partir des actions observées par opposition aux actions obtenues par abstraction. Ainsi, une action concrète est un symbole de Σ et une trace concrète est un mot de Σ^*.

Dans cette section, nous définissons formellement l'abstraction d'une trace d'exécution par rapport à un ensemble de behavior patterns. Nous considérons d'abord le cas d'un behavior pattern simple, puis le cas d'un behavior pattern régulier, et enfin le cas d'un ensemble de behavior patterns réguliers. Nous montrons que l'abstraction peut être vue comme un processus de réécriture de mots où chaque occurrence d'un behavior pattern est remplacée par un symbole particulier, propre au behavior pattern.

Abstraction par Rapport à un Behavior Pattern Simple

Soit $b \in \mathcal{B}$ un behavior pattern simple pour un certain behavior pattern $\mathcal{B}$. Identifier les occurrences du pattern b dans une trace $u \in \Sigma^*$ revient à rechercher b dans u.

Soit $\lambda \notin \Sigma$ un nouveau symbole dénotant l'abstraction de notre pattern b. L'abstraction de u par rapport au pattern $b \in \Sigma^*$ est définie par la réécriture de u avec le système de réécriture de mots R sur $(\Sigma \cup \{\lambda\})^*$, où R est composé d'une unique règle de réécriture : $R = \{b \to \lambda\}$.

Soit $\rightarrow_b$ la relation de réécriture induite par R sur $(\Sigma \cup \{\lambda\})^*$:

$$\begin{gathered} \forall u, u' \in (\Sigma \cup \{\lambda\})^*, \\ u \rightarrow_b u' \\ \Leftrightarrow \\ \exists u_1, u_2 \in (\Sigma \cup \{\lambda\})^*, \begin{cases} u = u_1 \cdot b \cdot u_2 \\ u' = u_1 \cdot \lambda \cdot u_2 \,. \end{cases} \end{gathered}$$

Exemple 23. Revenons à notre extrait du ver Allaple et supposons que nous souhaitions détecter le comportement de parcours des disques durs, défini précédemment. Alors on définit $b =$ `GetLogicalDriveStrings.GetDriveType.FindFirstFile` et $\lambda =$ `SCAN_DRIVES`.

La trace d'exécution de l'Exemple 22 est donc abstraite en :
`...SCAN_DRIVES.FindFirstFile.FindNextFile...`

Une trace u est complètement abstraite quand toute occurrence du pattern b dans u a été remplacée par le symbole abstrait λ. Ainsi, l'abstraction d'une trace correspond simplement à la normalisation de cette trace par rapport au SRS R. Notons que R n'est pas confluent en général et qu'une trace peut donc avoir plusieurs formes normales. On généralise de façon naturelle l'abstraction à un ensemble de traces.

Notons que l'on considère les formes normales au lieu des formes partiellement réduites. Autrement dit, on requiert que toute occurrence d'un pattern dans une trace soit finalement réécrite. Ainsi, le calcul du comportement de haut niveau exhibé par une trace donnée est maximal et la détection est donc plus précise que dans le cas d'une abstraction partielle. Par exemple, si l'on cherche le comportement "A suivi de B, sans C entre les deux" dans la trace acb, la forme partiellement réécrite AcB produirait un faux positif , alors que la forme normale ACB permet d'établir que cette trace n'exhibe pas comportement cherché.

Abstraction par Rapport à un Behavior Pattern Régulier

Lorsqu'une fonctionnalité peut être réalisée de plusieurs manières, un auteur de malware peut chercher à modifier légèrement un malware connu de façon à ce qu'il échappe à la détection. On souhaite donc abstraire de la même manière les différentes occurrences du behavior pattern décrivant une fonctionnalité donnée. Par exemple, la création d'un fichier peut être réalisée à l'aide différentes séquences d'appels de librairie et on veut pouvoir abstraire toutes ces séquences en le même symbole abstrait λ. On considère donc le cas de l'abstraction par rapport à un behavior pattern défini comme un langage régulier.

L'abstraction consiste alors à réécrire un ensemble de traces à l'aide d'un système de règles transformant tout behavior pattern simple d'un behavior pattern régulier $\mathcal{B}$ en un symbole λ. Étant donné que $\mathcal{B}$ peut être infini, on considère un SRS régulier $(\Sigma, V, S, \rightarrow)$ reconnaissant $\mathcal{B}$ et on définit la relation $\rightarrow_{\mathcal{B}}$ comme la relation de réécriture induite par $\rightarrow \cup \{S \rightarrow \lambda\}$ sur $(\Sigma \cup \{\lambda\} \cup V)^*$.

Exemple 24. On peut étendre le précédent behavior pattern `SCAN_DRIVES` de façon à prendre en compte des appels de librairie alternatifs, comme `FindFirstFileEx` au lieu de `FindFirstFile` :
`SCAN_DRIVES = GetLogicalDriveStrings.GetDriveType.(FindFirstFile+FindFirstFileEx).`
L'abstraction est alors réalisée par le SRS suivant :

$$\begin{aligned} \texttt{FindFirstFile} &\rightarrow A \\ \texttt{FindFirstFileEx} &\rightarrow A \\ \texttt{GetDriveType}.A &\rightarrow B \\ \texttt{GetLogicalDriveStrings}.B &\rightarrow \texttt{SCAN_DRIVES}\,. \end{aligned}$$

On définit ensuite l'abstraction d'un ensemble de traces par rapport à un behavior pattern régulier.

Définition 25 (Ensemble de traces abstrait)**.** Soit $\mathcal{B}$ un behavior pattern régulier, et $\rightarrow_{\mathcal{B}}$ sa relation de réécriture associée. La *forme abstraite* d'un ensemble de traces L par rapport à $\mathcal{B}$, notée $L{\downarrow_{\mathcal{B}}}$, est définie par :

$$L{\downarrow_{\mathcal{B}}} = \{v \in (\Sigma \cup \{\lambda\} \cup V)^* \mid \exists u \in L,\ u \rightarrow_{\mathcal{B}}^* v \text{ et } \nexists w \in (\Sigma \cup \{\lambda\} \cup V)^*,\ v \rightarrow_{\mathcal{B}} w\}.$$

Le théorème suivant exprime que l'abstraction préserve la régularité, une caractéristique fondamentale de notre approche.

Théorème 26. *Soit $\mathcal{B}$ un behavior pattern régulier et L un ensemble de traces. Si L est régulier, alors $L{\downarrow_{\mathcal{B}}}$ l'est aussi.*

Démonstration. Soit $R_{\mathcal{B}} = (\Sigma, V, \rightarrow, S)$. Pour un langage L, on note $R_{\mathcal{B}}^*(L)$ l'ensemble des successeurs d'éléments de L par $\rightarrow_{\mathcal{B}}$:

$$R_{\mathcal{B}}^*(L) = \{u \in (\Sigma \cup \{\lambda\} \cup V)^* \mid \exists v \in L, v \rightarrow_{\mathcal{B}}^* u\}\,.$$

Alors, l'ensemble $L{\downarrow_{\mathcal{B}}}$ est par définition l'ensemble des mots de $R_{\mathcal{B}}^*(L)$ qui sont en forme normale par rapport à $\rightarrow_{\mathcal{B}}$. Notons $Irr(\rightarrow_{\mathcal{B}})$ l'ensemble des mots de $(\Sigma \cup \{\lambda\} \cup V)^*$ qui sont irréductibles par $\rightarrow_{\mathcal{B}}$. On a alors :

$$L{\downarrow_{\mathcal{B}}} = R_{\mathcal{B}}^*(L) \cap Irr(\rightarrow_{\mathcal{B}}).$$

On montre alors que les ensembles $R^*_{\mathcal{B}}(L)$ et $Irr(\rightarrow_{\mathcal{B}})$ sont réguliers lorsque L est régulier.

En effet, le système $\rightarrow \cup \{S \rightarrow \lambda\}$ définissant $\rightarrow_{\mathcal{B}}$ est monadique [102], autrement dit ses règles sont de la forme $\alpha \rightarrow \beta$ avec $|\alpha| > |\beta|$ et $|\beta| \leq 1$. Or un système de réécriture monadique préserve la régularité des langages, comme l'ont montré Book et Otto dans [25] : l'ensemble $R^*_{\mathcal{B}}(L)$ est donc bien régulier lorsque L est régulier.

Dans un deuxième temps, notons $lhs(\rightarrow_{\mathcal{B}})$ l'ensemble des membres gauches des règles de $\rightarrow \cup \{S \rightarrow \lambda\}$. Alors, un mot $u \in (\Sigma \cup \{\lambda\} \cup V)^*$ est irréductible par $\rightarrow_{\mathcal{B}}$ ssi aucun sous-mot de u n'est un mot de $lhs(\rightarrow_{\mathcal{B}})$. On en déduit :

$$Irr(\rightarrow_{\mathcal{B}}) = \overline{(\Sigma \cup \{\lambda\} \cup V)^* . lhs(\rightarrow_{\mathcal{B}}) . (\Sigma \cup \{\lambda\} \cup V)^*}.$$

Or l'ensemble $lhs(\rightarrow_{\mathcal{B}})$ est fini et donc régulier, donc $Irr(\rightarrow_{\mathcal{B}})$ est régulier aussi. □

Abstraction par Rapport à un Ensemble de Behavior Patterns Réguliers

Finalement, on généralise l'abstraction à un ensemble de behavior patterns. En effet, en pratique, un comportement suspect peut être exprimé comme une combinaison de plusieurs behavior patterns $\mathcal{B}_i$, chacun d'entre eux s'abstrayant en un symbole distinct λ_i. Ces symboles abstraits λ_i appartiennent à l'ensemble Γ des actions abstraites.

Soit $\mathcal{C} = \{\mathcal{B}_i \mid \mathcal{B}_i \subseteq \Sigma^*\}_{1 \leq i \leq n}$ un ensemble fini de behavior pattern réguliers respectivement associés à des symboles distincts $\lambda_i \in \Gamma$.

Une relation $\rightarrow_{\mathcal{B}_i}$ étant définie sur $(\Sigma \cup \Gamma \cup V_i)^*$, la relation $\rightarrow_{\mathcal{C}} = \bigcup_{1 \leq i \leq n} \rightarrow_{\mathcal{B}_i}$ est définie sur $(\Sigma \cup \Gamma \cup \bigcup_{1 \leq i \leq n} V_i)^*$. On étend l'abstraction d'un ensemble de traces à un ensemble de behavior pattern réguliers, et l'ensemble de traces L est désormais normalisé avec $\rightarrow_{\mathcal{C}}$ en l'ensemble de traces abstrait $L{\downarrow_{\mathcal{C}}}$.

Théorème 27. *Soit $\mathcal{C}$ un ensemble fini de behavior patterns réguliers. Si L est régulier, alors $L{\downarrow_{\mathcal{C}}}$ l'est aussi.*

Démonstration. Soit $R_{\mathcal{B}_i} = (\Sigma, V_i, \rightarrow_i, S_i)$.

Soit $V = \bigcup_{1 \leq i \leq n} V_i$ et S une nouvelle variable distincte des variables de V.

Soit $\rightarrow_{\mathcal{C}} = \bigcup_{1 \leq i \leq n} (\rightarrow_i \cup \{S_i \rightarrow \lambda_i\})$.

On applique alors la preuve du Théorème 26 au SRS régulier suivant :

$$R_{\mathcal{C}} = \left(\Sigma \cup \{\lambda_i\}_{1 \leq i \leq n}, V, \rightarrow_{\mathcal{C}}, S\right).$$

□

Projection de l'Ensemble de Traces Abstrait sur Γ

Une fois un ensemble de traces abstrait par rapport à un ensemble de behavior patterns, les détails qui n'ont pas servi à l'abstraction doivent être éliminés. Intuitivement en effet, une trace complètement abstraite est définie sur Γ, de façon à être comparée ensuite à des comportements abstraits de référence. Cette opération est réalisée en projetant l'ensemble de traces abstrait sur Γ.

Définition 28 (Ensemble de traces Γ-abstrait)**.** Soit $\mathcal{C}$ un ensemble de behavior patterns réguliers. Soit L l'ensemble de traces d'un programme p. L'ensemble $\widehat{L} = L{\downarrow_{\mathcal{C}}}|_{\Gamma}$ est appelé l'*ensemble de traces* Γ*-abstrait* du programme p par rapport à $\mathcal{C}$.

Une fois l'abstraction complète, les traces simplifiées décrivent les séquences de fonctionnalités réalisées et permettent donc une détection plus robuste qu'en l'absence d'abstraction. Elles représentent en effet plusieurs implantations du même comportement et permettent ainsi de se protéger contre des modifications dans l'implantation des fonctionnalités qui les composent.

On peut ensuite comparer l'ensemble de traces Γ-abstrait au comportement abstrait d'un malware connu ou à un comportement générique quelconque défini sur Γ.

Remarque 29. Le processus d'abstraction complet pourrait être répété, comme dans l'architecture en couches de Martignoni et al. [81]. Une première couche identifierait des behavior patterns définis en termes des actions observées (i.e. sur Σ). Une seconde couche rechercherait des behavior patterns définis en termes des behavior patterns de la première couche, et ainsi de suite.

À l'inverse, on peut réduire le modèle multicouche à un modèle à une couche, c'est-à-dire définir les behaviors patterns de la dernière couche directement en termes des actions observées : pour cela, il suffit de composer les SRS réguliers définissant les behavior patterns des différentes couches. Les patterns résultants restent alors bien réguliers sur Σ, et notre formalisme s'applique.

4.4 Abstraction d'Ensembles Réguliers de Traces

En pratique, pour manipuler les ensembles de traces, on travaille sur des langages réguliers représentés par des automates de traces, dont la construction

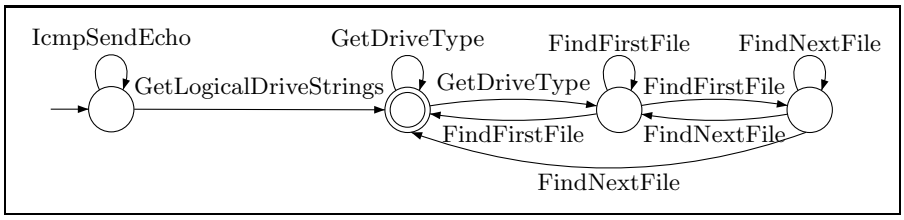

FIGURE 4.2: Automate de traces de l'extrait d'Allaple.a.

a été décrite dans le Chapitre 3. L'exemple suivant montre ainsi un automate de traces obtenu pour l'extrait du ver Allaple.a.

Exemple 30. Un automate de traces pour l'extrait du ver Allaple.a est représenté en Figure 4.2. Il représente en particulier le ping d'un hôte distant et le parcours des disques durs locaux.

On s'intéresse alors au calcul de l'ensemble de traces Γ-abstrait à partir d'un langage de traces régulier. Or on a vu avec le Théorème 27 que l'abstraction préservait la régularité de l'ensemble de traces. Le problème de l'abstraction d'un langage régulier de traces revient donc à calculer un automate reconnaissant la forme abstraite de ce langage. La construction de cet automate, que nous appelons automate de traces abstrait, est décrite dans les preuves des théorèmes suivants et utilise une méthode proposée par Esparza et al. [42]. Elle consiste à modifier l'automate de traces initial en ajoutant une nouvelle transition à chaque fois que le membre gauche d'une règle de réécriture est reconnu entre deux états de l'automate. Ensuite, l'automate obtenu est intersecté avec un automate reconnaissant les mots en forme normale par rapport à notre système de réécriture.

Ainsi, l'automate de traces abstrait peut être plus complexe que l'automate initial, comme le montre l'exemple du ver Allaple. L'abstraction de l'automate de traces par rapport aux patterns `SCAN_DRIVES` et `PING`, où `PING` = {`IcmpSendEcho`} décrit le ping d'un hôte distant, produit l'automate de la Figure 4.3.

Théorème 31. *Soit A un automate de traces à n états et $\mathcal{B}$ un behavior pattern reconnu par un SRS régulier $R_{\mathcal{B}}$.*

Alors un automate à $O(n)$ états reconnaissant $\mathcal{L}(A)\downarrow_{\mathcal{B}}\big|_{\Gamma}$ peut être construit en temps $O\left(n^3 \cdot |R_{\mathcal{B}}|^2\right)$ et en espace $O\left(n^2 \cdot |R_{\mathcal{B}}|\right)$.

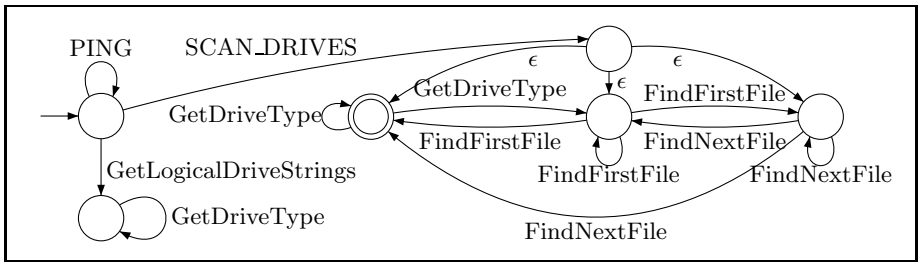

FIGURE 4.3: Automate de traces abstrait pour l'extrait d'Allaple.a.

Démonstration. D'après la preuve du Théorème 26, on a :

$$\mathcal{L}(A)\big\downarrow_{\mathcal{B}} = R^*_{\mathcal{B}}(\mathcal{L}(A)) \cap Irr(\rightarrow_{\mathcal{B}}).$$

Des méthodes de construction d'un automate reconnaissant $R^*_{\mathcal{B}}(L)$ sont proposées dans [42, 102]. En particulier, Esparza et al. proposent dans [42] une méthode pour construire un automate A' reconnaissant $R^*_{\mathcal{B}}(\mathcal{L}(A))$ en temps $O\left(|R_{\mathcal{B}}| \cdot n^3\right)$ et en espace $O\left(|R_{\mathcal{B}}| \cdot n^2\right)$. A' a le même nombre d'états que A. Par ailleurs, il est défini sur $\Sigma \cup \{\lambda\} \cup V$.

Construisons maintenant un automate reconnaissant l'ensemble $Irr(\rightarrow_{\mathcal{B}})$ afin de déterminer sa complexité.

Tout d'abord, nous construisons un automate déterministe reconnaissant $lhs(\rightarrow_{\mathcal{B}})$. Les règles de $\rightarrow \cup \{S \rightarrow \lambda\}$ sont de la forme $a \rightarrow B$, $aA \rightarrow B$ ou $S \rightarrow \lambda$, avec $a \in \Sigma$, $A \in V$, $B \in V$. Par conséquence, on partitionne $lhs(\rightarrow_{\mathcal{B}})$ en $\{S\} \cup \bigcup_{1 \leq i \leq n} a_i V_i$, avec $a_i \in \Sigma$, $V_i \subseteq V \cup \{\epsilon\}$, les a_i étant mutuellement distincts. On démarre avec un automate contenant un état initial i, un état final f et la transition $i \underset{S}{\rightarrow} f$. Pour chaque partition $a_i V_i$, on ajoute un nouvel état s_i et des transitions $i \underset{a_i}{\rightarrow} s_i$ et $s_i \underset{A}{\rightarrow} f$, avec $A \in V_i$ et $A \neq \epsilon$. Si $\epsilon \in V_i$, alors s_i est ajouté à l'ensemble des états finaux. Il existe au plus $|\Sigma|$ partitions (une pour chaque a_i) donc l'automate construit a $O(1)$ états et $O(|V|)$ transitions, ce qui prend un espace $O(|R_{\mathcal{B}}|)$. De plus, il y a $O(|R_{\mathcal{B}}|)$ règles de $\rightarrow$ et chaque règle doit être lue afin de partitionner $lhs(\rightarrow_{\mathcal{B}})$, donc cela prend aussi un temps $O(|R_{\mathcal{B}}|)$.

À partir de cet automate, on construit un automate complet, déterministe reconnaissant $\overline{Irr(\rightarrow_{\mathcal{B}})}$. Pour chaque état non final s_i provenant de la partition $a_i V_i$, on ajoute des transitions $s_i \underset{a}{\rightarrow} i$, pour chaque $a \in (\Sigma \cup \{\lambda\} \cup V) \setminus V_i$. Finalement, on ajoute une boucle sur $\Sigma \cup \{\lambda\} \cup V$ à chaque état final. Pour chaque

état, au plus $|\Sigma' \cup V|$ transitions ont été ajoutées, donc cette étape prend un temps $O(|V|) \subseteq O(|R_{\mathcal{B}}|)$ et un espace $O(|V|)$. L'automate résultant a toujours $O(1)$ états et $O(|V|)$ transitions. Finalement, en inversant l'ensemble des états finaux, on obtient son complément avec le même nombre d'états et de transitions, en temps constant et en espace $O(|V|)$.

Toute cette procédure construit donc, en temps et espace $O(|R_{\mathcal{B}}|)$, un automate A'' avec $O(1)$ états qui reconnaît $Irr(\rightarrow_{\mathcal{B}})$.

Maintenant, l'intersection sur l'alphabet $\Sigma \cup \{\lambda\} \cup V$ de deux automates ayant respectivement s et s' états produit un automate avec $s \cdot s'$ états, en un temps $O(s \cdot s' \cdot |V|^2 \cdot s')$ et un espace $O((s \cdot s')^2 \cdot |V|)$.

On en déduit, avec $s = n$ et $s' \in O(1)$, que l'intersection de A' et A'' prend un temps $O(n \cdot |V|^2)$ et un espace $O\left(n^2 \cdot |V|\right)$ et produit un automate avec $O(n)$ états et $O(n^2 \cdot |V|)$ transitions qui reconnaît $\mathcal{L}(A)\big\downarrow_{\mathcal{B}}$. Enfin, la projection sur Γ de cet automate est linéaire en son nombre de transitions, qui est dans $O(n^2 \cdot |V|)$.

En cumulant le tout, et en observant que $|V| = |R_{\mathcal{B}}|$, on obtient la complexité globale en temps suivante :

$$O\left(|R_{\mathcal{B}}| \cdot n^3\right) + O(|R_{\mathcal{B}}|) + O\left(n \cdot |R_{\mathcal{B}}|^2\right) + O(n^2 \cdot |R_{\mathcal{B}}|).$$

De même, la complexité globale en espace est :

$$O\left(|R_{\mathcal{B}}| \cdot n^2\right) + O(|R_{\mathcal{B}}|) + O\left(n^2 \cdot |R_{\mathcal{B}}|\right) + O(n^2 \cdot |R_{\mathcal{B}}|).$$

□

Théorème 32. *Soit A un automate de traces à n états et $\mathcal{C} = \{\mathcal{B}_i\}_{1 \leq i \leq n}$ un ensemble de behavior patterns reconnus par des SRS réguliers $\{R_{\mathcal{B}_i}\}_{1 \leq i \leq n}$. Soit $|\mathcal{C}| = \sum_{1 \leq i \leq n} |R_{\mathcal{B}_i}|$.*

Alors un automate à $O(n)$ états reconnaissant $\mathcal{L}(A)\big\downarrow_{\mathcal{C}}\big|_{\Gamma}$ peut être construit en temps $O\left(n^3 \cdot |\mathcal{C}|^2\right)$ et en espace $O\left(n^2 \cdot |\mathcal{C}|\right)$.

Démonstration. On procède comme dans la preuve du Théorème 27.

Soit $R_{\mathcal{B}_i} = (\Sigma, V_i, \rightarrow_i, S_i)$.

Soit $V = \bigcup_{1 \leq i \leq n} V_i$ et S une nouvelle variable distincte des variables de V.

Soit $\rightarrow_{\mathcal{C}} = \bigcup_{1 \leq i \leq n} (\rightarrow_i \cup \{S_i \rightarrow \lambda_i\})$.

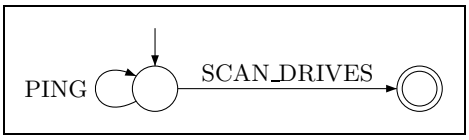

FIGURE 4.4: Automate Γ-abstrait pour l'extrait d'Allaple.a.

On applique la preuve du Théorème 31 au SRS régulier suivant :

$$R_{\mathcal{C}} = \left(\Sigma \cup \{\lambda_i\}_{1 \leq i \leq n}, V, \rightarrow_{\mathcal{C}}, S\right) .$$

Le nombre de variables de $R_{\mathcal{C}}$ est précisément $|\mathcal{C}| = \sum_{1 \leq i \leq n} |R_{\mathcal{B}_i}|$ donc cette preuve construit un automate à $O(n)$ états reconnaissant $\mathcal{L}(A)\big\downarrow_{\mathcal{C}}\big|_{\Gamma}$, en temps $O\left(n^3 \cdot |\mathcal{C}|^2\right)$ et en espace $O\left(n^2 \cdot |\mathcal{C}|\right)$. □

L'automate reconnaissant l'ensemble de traces Γ-abstrait d'Allaple.a, pour $\Gamma = \{\texttt{PING}, \texttt{SCAN_DRIVES}\}$, est représenté en Figure 4.4.

4.5 Détection de Comportements

En utilisant le formalisme d'abstraction que nous venons de définir (cf. Section 4.3), la détection comportementale consiste à calculer l'ensemble de traces abstrait d'un certain programme puis à le comparer à une base de comportements abstraits définis sur Γ. Ces comportements abstraits décrivent soit des comportements génériques, comme l'envoi de spam ou l'enregistrement des interactions clavier, soit des comportements de malwares spécifiques. Ils sont donc définis comme des ensembles de combinaisons particulières de fonctionnalités abstraites.

Définition 33. Un *comportement abstrait* sur Γ, ou *signature*, est un langage sur Γ.

Plus précisément, un comportement abstrait décrit des combinaisons de behavior patterns, éventuellement parsemées de behavior patterns additionnels qui ne sont pas pertinents dans ces combinaisons. Par exemple, on définit la signature du ver Allaple comme le langage régulier suivant, qui autorise explicitement l'entrelacement de behavior patterns ne correspondant pas au behavior pattern attendu :

$$\Gamma^* \cdot \texttt{LOCAL_COM_SERVER} \cdot (\Gamma \setminus \{\texttt{PING}\})^* \cdot \texttt{PING} \cdot$$
$$(\Gamma \setminus \{\texttt{NETBIOS_CONNECTION}\})^* \cdot \texttt{NETBIOS_CONNECTION} \cdot \Gamma^*.$$

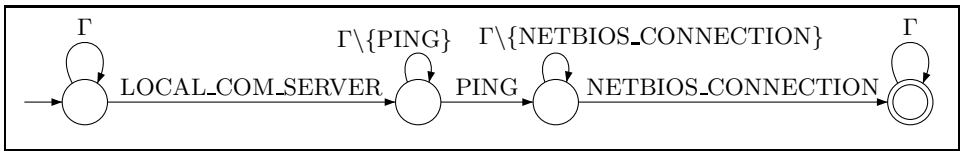

FIGURE 4.5: Signature d'Allaple.

L'automate représentant la signature d'Allaple est représenté en Figure 4.5. Notons que le pattern `SCAN_DRIVES`, qui est présent dans l'automate de traces Γ-abstrait de l'extrait d'Allaple.a, n'apparaît pas ici car la signature décrit un comportement discriminant commun à l'ensemble des échantillons d'Allaple.

Définition 34. Soit M un comportement abstrait sur Γ. Un programme p, avec un ensemble de traces Γ-abstrait $\widehat{L}$, *exhibe le comportement abstrait* M ssi : $\widehat{L} \cap M \neq \emptyset$.

Ainsi, p exhibe le comportement M ssi une de ses traces abstraites est dans M. Lorsque $\widehat{L}$ est représenté par un automate A et M par un automate A_M, le programme p exhibe le comportement M ssi : $\mathcal{L}(A) \cap \mathcal{L}(A_M) \neq \emptyset$.

Notre *base de comportements abstraits* est alors un ensemble $\mathcal{D}$ de comportement abstraits. Lorsque tous ces comportements abstraits sont réguliers, on représente la base de comportements abstraits par un automate $A_{\mathcal{D}}$ qui est l'union des automates A_M représentant les signatures. L'analyse comportementale du programme p a alors la complexité suivante.

Théorème 35. *Soit $\mathcal{D}$ un ensemble de comportements abstraits réguliers sur Γ, reconnu par un automate $A_{\mathcal{D}}$. Soit un programme p, avec un ensemble de traces Γ-abstrait reconnu par un automate A. Alors décider si p exhibe un comportement de $\mathcal{D}$ prend un temps : $O(|A_{\mathcal{D}}| \cdot |A|)$.*

Démonstration. Décider si p est malicieux par rapport à $\mathcal{D}$ revient à tester si le langage $\mathcal{L}(A_{\mathcal{D}}) \cap \mathcal{L}(A)$ est vide.

L'intersection des deux automates prend un temps $O(|A_{\mathcal{D}}| \cdot |A|)$ et produit un automate de taille $O(|A_{\mathcal{D}}| \cdot |A|)$. Puis tester si l'automate résultant reconnaît l'ensemble vide prend un temps linéaire en sa taille, d'où le résultat. □

Notons que notre technique d'analyse comportementale peut être efficacement adaptée à l'analyse en temps réel à partir de ces résultats, étant donné que toutes les constructions (Γ-abstraction et intersection d'automates) sont incrémentales.

Remarque 36. L'infection pourrait être définie de façon plus intuitive et plus générale de la manière suivante : les behavior patterns, au lieu de représenter les blocs de base d'un comportement abstrait, pourraient représenter directement un comportement abstrait, qui serait alors défini sur Σ. Un programme serait alors infecté ssi une de ses traces au moins contenait une occurrence d'un behavior pattern. Mais on perdrait alors le pouvoir expressif de l'abstraction et la robustesse de notre modèle de détection. D'un autre côté toutefois, la détection serait alors réalisée uniquement par normalisation de traces.

4.6 Expérimentations

Pour valider notre formalisme, nous avons implanté et testé les techniques décrites sur des échantillons de programmes malicieux. Ces échantillons ont été collectés sur un pot de miel local au Laboratoire de Haute Sécurité du Loria [1] (les échantillons sont identifiés par Kaspersky Antivirus) et dans des bases de données publiques, comme Offensive Computing [2]. Nous avons réalisé un outil construisant l'automate de traces d'un programme à partir d'un ensemble de traces capturées par analyse dynamique (cf. Chapitre 3), permettant de l'abstraire par rapport à un ensemble de behavior patterns prédéfinis et de comparer l'automate de traces abstrait résultant à une base de comportements malicieux.

Behavior Patterns

Les behavior patterns sont définis par observation de traces d'exécution malicieuses, à partir desquelles on extrait des séquences courtes susceptibles de composer un comportement suspect. Ces patterns définissent souvent des interactions habituelles du programme entre le système et le réseau. Une fois extraite, une séquence permet de définir un nouveau behavior pattern ou d'étendre un behavior pattern existant. Rappelons que la notion de trace d'exécution étendue a été définie dans le Chapitre 2 pour représenter dans les traces des valeurs spécifiques des arguments, comme le fait que le fichier ouvert est un fichier système ou que la clé de registre modifiée est la clé `Run` permettant de lancer le programme à chaque ouverture de session. Cela permet de définir des fonctionnalités précises comme la fonctionnalité de modification d'un fichier système.

Environ 40 behavior patterns ont été construits, décrivant :

1. http://lhs.loria.fr
2. http://www.offensivecomputing.net

- une écriture de fichiers systèmes, qui consiste à ouvrir un fichier dans un répertoire système et à écrire dans ce fichier, ou à copier un fichier dans un répertoire système ;
- un enregistrement dans la base de registre de Windows pour être automatiquement exécuté au démarrage d'une session (comportement de persistance), qui consiste à ajouter une sous-clé dans la clé de registre `Run` ou à modifier la sous-clé `AppInit_DLLs` ;
- une installation d'un service (lancé automatiquement : comportement de persistance) ;
- un parcours des disques et des fichiers, qui consiste à récupérer la liste des dossiers et à scanner leurs fichiers ;
- une communication sur IRC, qui consiste à envoyer des messages IRC sur une connexion réseau, tels que le message obligatoire "`NICK pseudo`" ;
- une recherche d'antivirus installés, qui consiste à chercher des entrées de registre et des services systèmes spécifiques ;
- une vérification de l'existence d'un mutex ou d'un évènement ;
- une mise en place d'une hook système, une utilisation d'astuces antidebug, une énumération de processus, une demande ou une vérification de droits d'accès, par exemple pour vérifier que l'utilisateur est bien administrateur de sa machine, etc.

Signatures

La base de signatures est une collection de comportements de haut niveau composées de deux types de comportements : des comportements provenant de malwares connus et des comportements génériques, ne représentant pas un malware spécifique. Dans le cas d'un malware connu, son automate de traces Γ-abstrait est d'abord construit puis il est modifié pour en éliminer toute séquences d'actions innocente et ne conserver que les séquences effectuant effectivement une action préjudiciable. À partir de l'analyse des automates de traces Γ-abstraits de malwares connus, des comportements génériques peuvent être définis en identifiant des séquences de fonctionnalités communes à plusieurs malwares et caractéristiques d'un comportement malicieux.

Mise en Œuvre

Deux outils ont été développés. Le premier, présenté dans le Chapitre 3, est un outil de capture qui permet de capturer des traces d'exécution par analyse dynamique. Il a été développé en C (environ 6000 lignes de code) et repose sur l'outil d'instrumentation binaire Pin [8]. Comme nous l'avons expli-

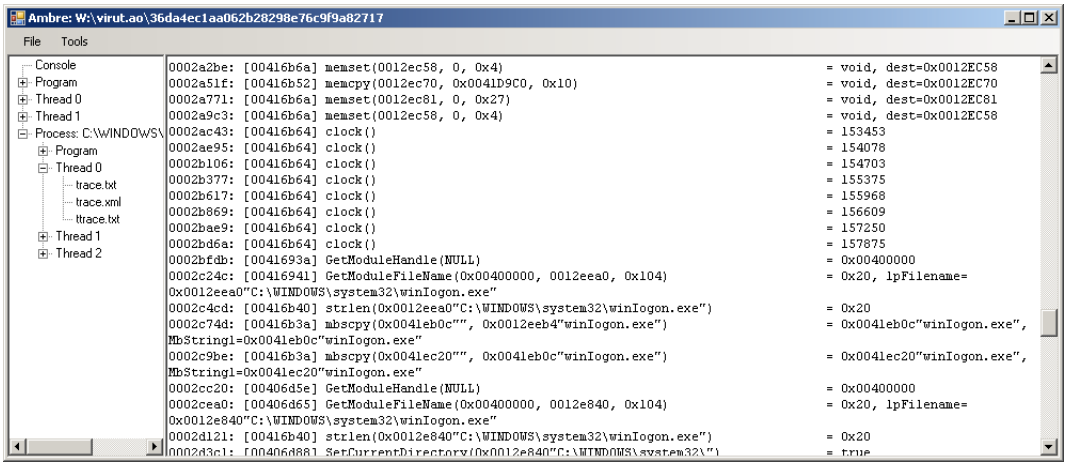

FIGURE 4.6: Outil de capture - Capture de traces.

qué en Section 3.1, les traces capturées doivent être suffisamment expressives pour permettre ensuite la construction d'un automate de traces. La Figure 4.6 illustre le type de traces capturées dans le cas du virus Virut.ao. Chaque appel de librairie est décrit par un compteur, son adresse d'appel, son nom, la valeur de ses arguments d'entrée, sa valeur de retour et la valeur de ses arguments de sortie. De plus, comme on le voit sur la figure, l'outil de capture permet de capturer les traces des threads et des processus enfants.

Le deuxième outil est un outil d'analyse développé en C# (environ 9000 lignes de code) qui permet de construire un automate de traces à partir d'un ensemble de traces capturées puis de soumettre cet automate de traces à une base de comportements de haut niveau en :

- abstrayant en forme normale l'automate de traces par rapport à un ensemble de behavior patterns puis en le projetant sur l'alphabet Γ ;
- faisant appel aux algorithmes de la librairie OPENFST pour réaliser un test du vide de l'intersection de l'automate de traces Γ-abstrait ainsi construit et de l'automate représentant la base de signatures.

Les automates sont manipulés à l'aide de la librairie OPENFST [7].

La Figure 4.7 montre l'automate de traces construit par l'outil d'analyse à partir d'un ensemble de traces capturées à l'aide de l'outil de capture.

La Figure 4.8 montre un exemple de behavior pattern par rapport auquel est effectuée l'abstraction.

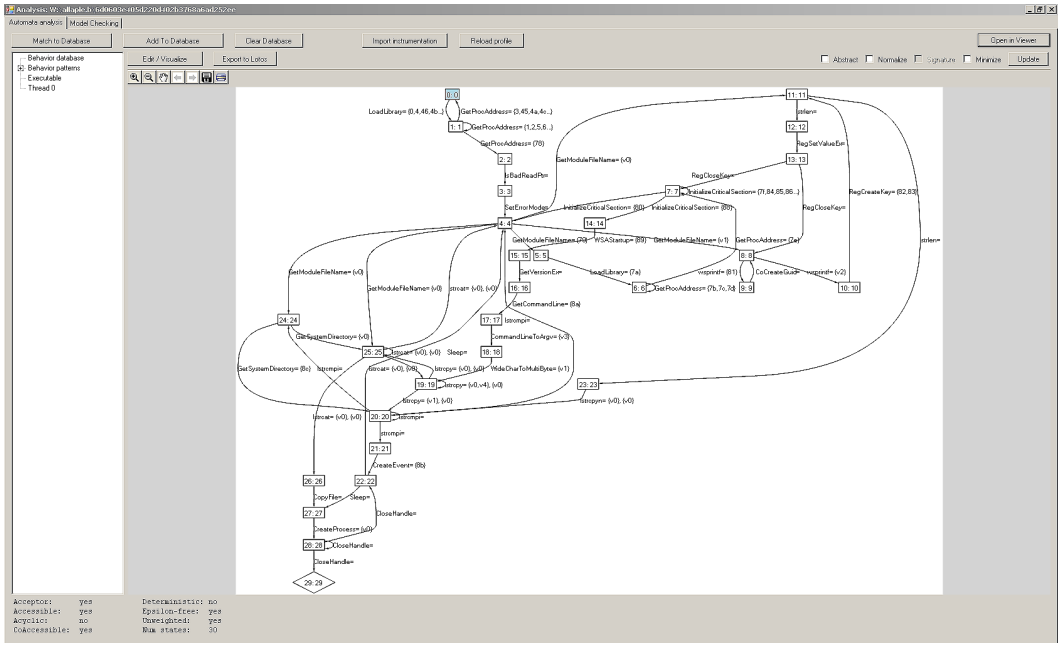

FIGURE 4.7: Outil d'analyse - Construction de l'automate de traces.

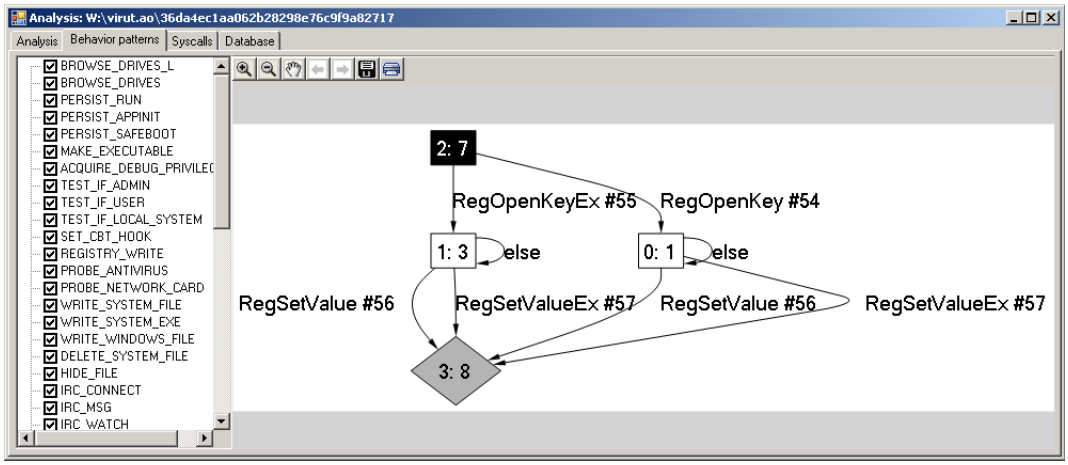

FIGURE 4.8: Outil d'analyse - Définition des behavior patterns.

Résultats

Trois scénarios d'expérimentations ont été mis en place.

Dans une premier scénario, nous avons défini des signatures pour des familles de malware spécifiques, en analysant et en Γ-abstrayant quelques échantillons de cette famille. Des échantillons de variantes distinctes de chaque famille ont ensuite été comparées à ces signatures : plusieurs de ces variantes étaient nouvelles, autrement dit elles n'avaient pas été considérées lors de la définition des signatures. Cela nous a permis de tester l'applicabilité de notre approche et sa robustesse face à la mutation.

Dans un second scénario, une signature plus générale a été définie pour un comportement malicieux courant rencontré au sein de plusieurs familles de malwares. Plusieurs échantillons malicieux ont ensuite été testés, afin de découvrir quels échantillons exhibaient ce comportement. Cela nous a permis de tester le pouvoir expressif de l'abstraction de comportements.

Dans un troisième scénario, des applications saines ont été testées, afin de s'assurer que le taux de faux positifs était bas.

Nous avons testé ces trois scénarios sur des familles connues de malware, parmi lesquelles Allaple, Virut, Agent, Rbot, Afcore et Mimail. En particulier, notre pot de miel montre que Allaple, Virut et Agent font actuellement partie des vers les plus actifs, ce qui rend leur analyse particulièrement pertinente. La plupart des échantillons ont été comparés avec succès aux signatures définies.

Les échecs rencontrés étaient dus à des limitations techniques de notre implantation ou de l'analyse dynamique. Par exemple, plusieurs malwares injectent leur code dans des applications en cours d'exécution afin de s'exécuter dans le contexte de ces applications : notre outil de capture ne permettait pas, en l'état, de suivre l'exécution du code injecté. De même, certains malwares utilisent les services Windows pour s'exécuter avant l'ouverture d'une session : notre outil de capture ne permettait pas non plus de capturer les traces d'exécution de services Windows. Enfin, les traces capturées par analyse dynamique n'étaient pas toujours suffisamment expressives, l'exécution pouvant s'arrêter prématurément pour plusieurs raisons : la configuration de l'ordinateur ne correspond pas à la configuration attendue, l'instrumentation du programme ou son exécution dans une machine virtuelle a été détectée, la connexion à un serveur Web échoue en raison de l'indisponibilité du serveur, etc. Dans ce cas, l'analyse statique peut être utilisée pour contourner ces limitations.

La Figure 4.9 montre l'automate de traces d'un thread du ver d'emails Agent.ah. Il est restreint aux appels de fonction utilisés dans les behavior patterns définis. L'automate de traces a ensuite été abstrait par rapport à notre ensemble de 40 behavior patterns. Trois de ces behavior patterns ont été

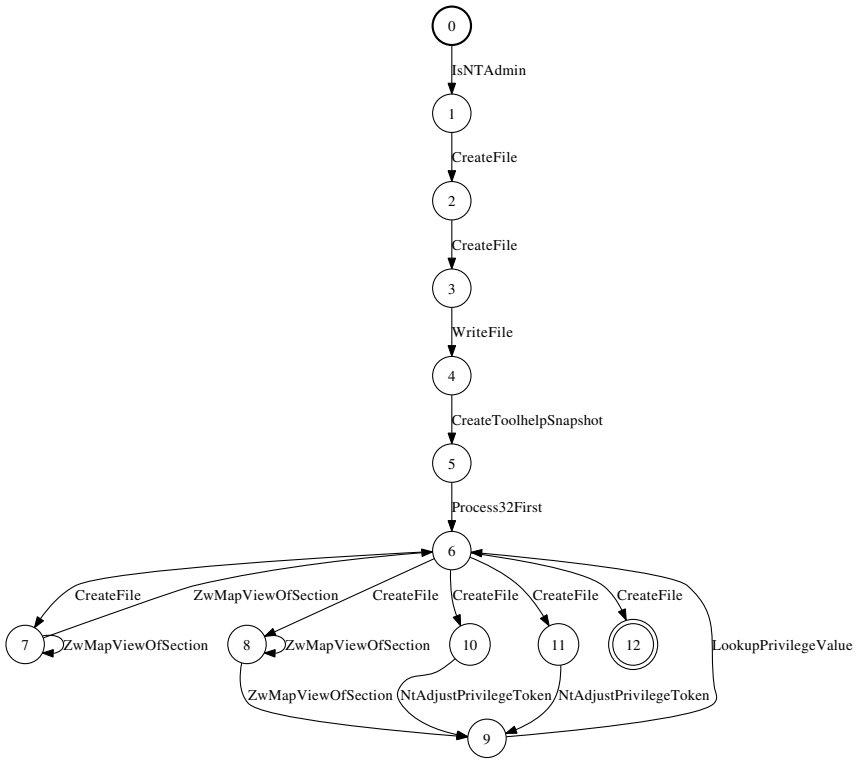

FIGURE 4.9: Automate de traces partiel de Agent.ah.

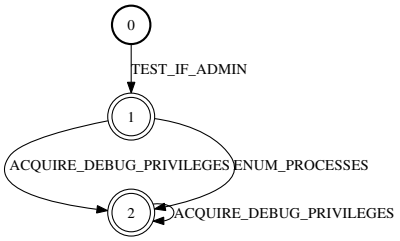

FIGURE 4.10: Automate de traces Γ-abstrait de Agent.ah.

reconnus et abstraits :

- `TEST_IF_ADMIN` : Vérifie si l'utilisateur a des droits d'administrateur.
  `TEST_IF_ADMIN = IsNtAdmin + (OpenThreadToken + OpenProcessToken) . AllocateAndInitializeSid [nSubAuthorityCount = 2, dwSubAuthority0 =`
  `0x20, dwSubAuthority1 = 0x220] . AccessCheck`
- `ACQUIRE_DEBUG_PRIVILEGES` : Demande au système d'exploitation des privilèges de déboguage, afin d'accéder à l'espace mémoire des processus en cours d'exécution.
  `ACQUIRE_DEBUG_PRIVILEGES = LookupPrivilegeValue [lpName = ''SeDebug Privilege''] . (NtAdjustPrivilegesToken [NewState->Privileges[0]. Attributes = 2] + AdjustTokenPrivileges [NewState->Privileges[0]. Attributes = 2])`
- `ENUM_PROCESSES` : Énumère les processus en cours d'exécution.
  `ENUM_PROCESSES = EnumProcesses + CreateToolhelp32Snapshot [dwFlags &`
  `2 != 0] . Process32First`

L'automate de traces Γ-abstrait d'Agent.ah est montré en Figure 4.10.

4.7 Conclusion

Dans ce chapitre, nous avons présenté une première approche pour la détection de logiciels malveillants à partir de représentations abstraites des comportements suspects. Le programme à analyser, représenté par un automate de traces, est abstrait, à l'aide d'un système de réécriture de mots, par rapport à des behavior patterns élémentaires définis comme des langages réguliers. L'ensemble de traces ainsi abstrait est ensuite comparé à une base de comportements abstraits, définis comme des ensembles de combinaisons suspectes de ces behavior patterns.

L'abstraction est une notion clé de notre approche. En travaillant sur une forme abstraite du comportement d'un programme et des comportements suspects à détecter, notre méthode d'analyse de comportements est indépendante de l'implantation du programme, gère les comportements similaires de façon générique et est robuste par rapport aux variantes existantes et à venir. La force de notre technique repose aussi dans le fait que les comportements suspects abstraits sont des combinaisons de fonctionnalités élémentaires. L'expression de ces comportements devient alors flexible et aisément modifiable si besoin est. Il en est de même des behavior patterns puisqu'ils décrivent des fonctionnalités de base et non des fonctionnalités complexes.

Notons que notre formalisme peut également s'appliquer à d'autres scénarios que la détection, comme l'analyse de similarité de programmes malicieux.

Comme pour la détection, l'utilisation de représentations abstraites des programmes rend l'analyse résiliente à toute modification mineure du code qui n'a pas d'impact sur les fonctionnalités mises en œuvre.

Chapitre 5

Abstraction de Comportements par Réécriture de Termes

Nous considérons maintenant une formalisation de l'abstraction basée sur des termes. En effet, la représentation des traces d'exécution par des termes permet de représenter le flux de données comme on l'a vu dans le Chapitre 2. Ainsi, en définissant directement l'abstraction de traces sur une algèbre de termes, on peut exprimer des contraintes sur le flux de données, dans la définition des behavior patterns comme dans la définition des comportements de haut niveau à détecter. Dans ce chapitre, nous proposons donc un formalisme de détection de comportements de haut niveau qui repose sur la réécriture de termes.

De plus, dans le formalisme du chapitre précédent, les occurrences entrelacées de behavior patterns n'étaient pas prises en compte exhaustivement étant donné que, lors de l'abstraction de la première occurrence, la seconde occurrence était partiellement consommée et ne pouvait donc plus être identifiée. Nous adaptons donc notre mécanisme pour que les behavior patterns originels soient conservés dans l'abstraction.

Enfin, on constate que, les behavior patterns et les comportements de haut niveau à détecter pouvant être définis naturellement à l'aide de formules de logique temporelle, l'abstraction et la détection s'apparentent à du model checking [75]. En effet, l'abstraction d'une trace peut être interprétée comme la validation sur cette trace d'une formule de logique temporelle décrivant un behavior pattern et, en cas de succès, c'est-à-dire si une occurrence du behavior pattern est détectée, comme le marquage de cette occurrence dans la trace. La détection peut quant à elle être interprétée comme la validation sur une trace abstraite de la formule de logique temporelle décrivant le comportement

recherché. Notre approche, lors de sa mise en pratique, pourra dès lors permettre d'utiliser les outils existants du model checking pour l'abstraction et la détection. Aussi, dans ce chapitre, nous rendons explicites les mécanismes de model checking sous-jacents à notre formalisme d'analyse comportementale par abstraction.

Plus précisément, nous cherchons à voir si un programme exhibe un comportement de haut niveau, décrit par une formule de logique temporelle du premier ordre. La détection est réalisée en deux étapes. Dans un premier temps, les traces du programme sont abstraites de façon à révéler les séquences de fonctionnalités de haut niveau qu'elles réalisent, définies elles aussi par des formules de logique temporelle du premier ordre. Dans un deuxième temps, les traces abstraites sont comparées avec la formule de logique temporelle décrivant le comportement, en utilisant les techniques usuelles de model checking.

L'abstraction consiste à identifier et marquer les fonctionnalités de haut niveau dans les traces du programme. À la différence du formalisme du chapitre précédent, l'abstraction consiste non pas à remplacer les occurrences des fonctionnalités par des actions abstraites mais à marquer ces occurrences tout en les gardant, par insertion d'une action abstraite. Ainsi, l'intégrité des traces est préservée et deux occurrences apparaissant entrelacées dans une trace pourront être toutes deux identifiées.

Observons que, bien que notre mécanisme de détection s'appuie sur le model checking, pour lequel il existe déjà de nombreux cadres d'application et de nombreux outils [59, 45, 29], la spécificité de notre approche est que la vérification n'est pas appliquée à l'ensemble de traces associé au programme mais à l'ensemble de formes abstraites de ces traces, ensemble qui n'est en général pas calculable. Par conséquent, nous identifions une propriété des comportements de haut niveau en pratique qui nous permet d'approximer cet ensemble, de façon correcte et complète vis à vis de la détection, et d'appliquer ensuite les techniques de vérification classiques.

Notre mécanisme peut là-encore être utilisé dans deux scénarios :

- *Détection de comportements donnés* : on cherche à découvrir dans un programme si une de ses traces d'exécution exhibe un comportement de haut niveau donné. Ce comportement est défini en termes de fonctionnalités abstraites.
- *Analyse de programmes* : l'abstraction fournit une représentation de haut niveau du comportement du programme qui peut être utilisée pour une analyse manuelle ou pour une analyse de similarité du programme avec des codes malveillants connus. En outre, elle peut également être utili-

sée pour découvrir automatiquement dans les traces du programme des séquences de fonctionnalités de haut niveau avec leur flux de données.

5.1 Définitions

Termes

Pour $f \in \mathcal{F}$ d'arité $n \in \mathbb{N}$, on note $f(\overline{x})$ pour $f(x_1, \ldots, x_n)$, où $x_1, \ldots, x_n$ sont des variables.

Une substitution close (ou instanciation) sur un ensemble fini X de variables S-sortées est une application $\sigma : X \to T(\mathcal{F})$ telle que : $\forall s \in S, \forall x \in X_s$, $\sigma(x) \in T_s(\mathcal{F})$. L'instanciation σ peut être étendue de manière naturelle à une application $T(\mathcal{F}, X) \to T(\mathcal{F})$ de telle façon que :

$$\begin{gathered}\forall f(t_1, \ldots, t_n) \in T(\mathcal{F}, X), \\ \sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n)).\end{gathered}$$

Par convention, on note $t\sigma$ ou $\sigma(t)$ l'application d'une instanciation σ à un terme $t \in T(\mathcal{F}, X)$ et $L\sigma$ l'application de σ à un ensemble de termes $L \subseteq T(\mathcal{F}, X)$. L'*ensemble des instanciations sur* X est noté $Inst_X$.

On partitionne $\mathcal{F}_a$ en un ensemble Σ de symboles permettant de définir les actions concrètes du programme (c'est-à-dire les actions telles qu'elles sont observées à l'exécution du programme et apparaissent dans l'état originel de la trace) et un ensemble Γ de symboles permettant de définir les actions abstraites (c'est-à-dire identifiant les fonctionnalités qui ont été abstraites). On appelle alors trace concrète (resp. concrète) une trace composée uniquement d'actions concrètes (resp. abstraites). Pour représenter l'ensemble des traces et des actions concrètes (resp. abstraites), on utilise l'alphabet $\mathcal{F}_\Sigma = \mathcal{F} \setminus \Gamma$ (resp. $\mathcal{F}_\Gamma = \mathcal{F} \setminus \Sigma$). Ainsi, $T_{Trace}(\mathcal{F}_\Sigma)$ représente l'ensemble des traces concrètes. De même, $T_{Action}(\mathcal{F}_\Gamma)$ représente l'ensemble des actions abstraites. Plus généralement, pour tout alphabet $\Omega \subseteq \mathcal{F}_a$, on définit $\mathcal{F}_\Omega = \mathcal{F}_t \cup \mathcal{F}_d \cup \Omega$.

On définit de façon naturelle la concaténation $t \cdot t'$ de deux termes t et t' de $T_{Trace}(\mathcal{F}, X)$, où X est un ensemble de variables S-sortées et $t \notin X$, par :

$$t \cdot t' = t\left[t'\right]_p$$

où p est la position de ϵ dans t, i.e. $t|_p = \epsilon$.

De même, si $\Sigma' \subseteq \mathcal{F}_a$ est un ensemble de symboles d'actions et si X est un ensemble de variables de sorte $Data$, on définit la projection sur Σ' d'un terme

t de $T_{Trace}(\mathcal{F}, X)$, notée $\pi_{\Sigma'}(t)$ ou $t|_{\Sigma'}$, par :

$$\begin{aligned} \pi_{\Sigma'}(\epsilon) = & \ \epsilon \\ \pi_{\Sigma'}(b \cdot u) = & \begin{cases} b \cdot \pi_{\Sigma'}(u) & \text{if } b \in T_{Action}(\mathcal{F}_{\Sigma'}, X) \\ \pi_{\Sigma'}(u) & \text{otherwise} \end{cases} \end{aligned}$$

où $b \in T_{Action}(\mathcal{F}, X)$ et $u \in T_{Trace}(\mathcal{F}, X)$.

La projection et la concaténation sont étendues de façon naturelle à un ensemble de termes de sorte *Trace*. On étend aussi la concaténation à $2^{T_{Trace}(\mathcal{F},X)} \times 2^{T_{Trace}(\mathcal{F},X)}$, avec $L \cdot L' = \{t \cdot t' \mid t \in L, t' \in L'\}$, et à $2^{T_{Trace}(\mathcal{F},X)} \times T_{Action}(\mathcal{F}, X)$, avec $L \cdot a = L \cdot \{a \cdot \epsilon\}$.

Relations Binaires

Soit R une relation binaire sur $T(\mathcal{F}, X)$. On définit $R^* = \bigcup_{i \geq 0} R^i$ et, pour tout $n \in \mathbb{N}$, $R^{\leq n} = \bigcup_{0 \leq i \leq n} R^i$.

On appelle ensemble des descendants d'un terme t (resp. d'un ensemble de termes L) l'ensemble $R^*(t)$ (resp. l'ensemble $R^*(L)$).

Soit $n \in \mathbb{N}$. On appelle ensemble des descendants d'un terme t (resp. d'un ensemble de termes L) jusqu'à l'ordre n l'ensemble $R^{\leq n}(t)$ (resp. l'ensemble $R^{\leq n}(L)$).

First-Order LTL (FOLTL) Temporal Logic

Nous considérons une extension de la logique temporelle LTL à des prédicats atomique avec arité dont les arguments sont définis sur un domaine fini. Une telle extension a été étudiée pour le μ-calcul dans [52, 82] et est un cas particulier de la logique temporelle du premier ordre FOLTL définie par Kröger et Merz dans [75].

Dans notre cas, on travaille sur le domaine des termes de sorte *Data* et on associe à chaque symbole d'action de $\Sigma \cup \Gamma$ un symbole de prédicat atomique de même arité. Par souci de clarté, on écrira $AP = \Sigma \cup \Gamma$ pour définir l'ensemble des symboles de prédicats atomiques sur lequel on travaille.

Soit X un ensemble fini de variables de sorte *Data*. Définissons l'ensemble (fini) des formules atomiques : $AF(X) = \{f(d_1, \ldots, d_n) \mid f \in AP, d_1, \ldots, d_n \in T_{Data}(\mathcal{F}, X)\}$. En particulier, l'ensemble $AF(\emptyset)$ désigne l'ensemble des formules atomiques closes. Contrairement à Kröger et Merz [75], nous n'incluons pas l'égalité entre termes de $T_{Data}(\mathcal{F}, X)$ dans les formules atomiques.

La logique FOLTL est alors définie comme une extension de la logique temporelle LTL de la façon suivante :

- $\top$ (true) et $\bot$ (false) sont des formules FOLTL ;
- Si $a \in AF(X)$, alors a est une formule FOLTL ;
- Si φ_1 et φ_2 sont des formules FOLTL, alors : $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $\mathbf{X}\varphi_1$ ("next time"), $\mathbf{F}\varphi_1$ ("eventually" ou "in the future") et $\varphi_1 \,\mathbf{U}\, \varphi_2$ ("until") sont des formules FOLTL.
- Si φ est une formule FOLTL et $Y \subseteq X$ est un ensemble de variables (de sorte *Data*), alors : $\exists Y.\varphi$ et $\forall Y.\varphi$ sont des formules FOLTL, où, comme c'est le cas usuellement : $\forall Y.\varphi \equiv \neg\exists Y.\neg\varphi$.

Une formule FOLTL est dite close si elle ne contient pas de variable libre, autrement dit si toute variable est liée à un quantificateur.

Observons que la quantification se faisant sur des variables à domaine fini *Data*, toute formule FOLTL close est équivalente à une formule FOLTL sans quantificateur.

On notera $\varphi_1 \odot \varphi_2$ pour $\varphi_1 \wedge \mathbf{XF}\varphi_2$, qui décrit la formule "φ_1 est vraie au temps 0 puis, un peu plus tard, φ_2 est vraie".

Des exemples de formules FOLTL closes sont :

- $fopen(1) \odot fwrite(1,2)$;
- $\exists x, y.\, fopen\,(x) \odot fwrite\,(x,y)$;
- $\exists x, y.\, fopen\,(x) \wedge \neg fclose\,(x)\, \mathbf{U} fwrite\,(x,y)$;
- $\exists x.\, fopen\,(x) \wedge (\forall y.\, \neg fwrite\,(x,y))\, \mathbf{U} fclose\,(x)$.

Soit $Y \subseteq X$ un ensemble de variables (de sorte *Data*) et $\sigma \in Inst_Y$ une instanciation sur Y. On définit de façon naturelle l'application de σ à une formule FOLTL φ par la formule $\varphi\sigma$ où toute variable libre x de φ qui est dans Y a été remplacée par sa valeur $\sigma\,(x)$.

Soit un alphabet A. On note A^ω l'ensemble des mots infinis sur A : $A^\omega = \{a_1 \cdot a_2 \cdots \mid \forall i,\, a_i \in A\}$.

Une formule FOLTL close est validée sur des séquences infinies d'ensembles de formules atomiques, notées $\xi = (\xi_0, \xi_1, \ldots) \in \left(2^{AF(\emptyset)}\right)^\omega$. On note ξ^i la séquence $(\xi_i, \xi_{i+1}, \ldots)$. $\xi \models \varphi$ (ξ valide φ) est défini par :

- $\xi \models \top$;
- $\xi \models a$, avec $a \in AF(\emptyset)$, ssi $a \in \xi_0$;
- $\xi \models \neg\varphi$ ssi $\xi \not\models \varphi$;
- $\xi \models \varphi_1 \wedge \varphi_2$ ssi $\xi \models \varphi_1$ et $\xi \models \varphi_2$;
- $\xi \models \varphi_1 \vee \varphi_2$ ssi $\xi \models \varphi_1$ ou $\xi \models \varphi_2$;
- $\xi \models \mathbf{X}\varphi$ ssi $\xi^1 \models \varphi$;
- $\xi \models \mathbf{F}\varphi$ ssi pour un certain $i \geq 0$, $\xi^i \models \varphi$;
- $\xi \models \varphi_1 \mathbf{U} \varphi_2$ ssi pour un certain $i \geq 0$, $\xi^i \models \varphi_2$ et, pour tout $j \in [0..i-1]$, $\xi^j \models \varphi_1$;
- $\xi \models \exists Y.\varphi$ ssi il existe une instanciation $\sigma \in Inst_Y$ telle que $\xi \models \varphi\sigma$.

Dans notre contexte, une formule est validée sur des traces de $T_{Trace}\,(\mathcal{F})$ iden-

tifiées à des séquences d'ensembles singletons de formules atomiques. Une trace finie $t = a_0 \cdots a_n$ est identifiée à la séquence infinie d'ensembles de formules atomiques $\xi_t = (\{a_0\}, \ldots, \{a_n\}, \{\}, \{\}, \ldots)$, et t valide φ, noté $t \models \varphi$, ssi $\xi_t \models \varphi$.

Notons que nous ne considérons pas de quantification sur les chemins (par exemple en considérant la logique CTL* plutôt que LTL) car les propriétés que nous cherchons à valider en analyse comportementale décrivent la réalisation d'un certain comportement (sur un chemin quelconque) et non la réalisation d'un certain comportement sur l'ensemble des chemins possibles. Toutefois, une telle extension de la logique temporelle est plus expressive et en ce sens pourrait constituer une extension intéressante de notre formalisme.

Transducteur d'Arbres

Soit X un ensemble de variables. Un transducteur d'arbres (descendant) [34] est un quintuplet $\tau = (\mathcal{F}, \mathcal{F}', Q, q_0, \Delta)$ où $\mathcal{F}$ est un ensemble fini de symboles d'entrée, $\mathcal{F}'$ est un ensemble fini de symboles de sortie, Q est un ensemble fini d'états unaires, $q_0 \in Q$ est un état initial, Δ est un ensemble fini de règles de transduction de la forme :

$$\begin{array}{cc} q\left(f\left(x_1, \ldots, x_n\right)\right) \to u & \\ \text{ou} & \\ q\left(x_1\right) \to u & \qquad (\epsilon\text{-transition}) \end{array}$$

où $q \in Q$, $n \in \mathbb{N}$, $f \in \mathcal{F}$ est un symbole d'arité n, $x_1, \ldots, x_n$ sont des variables distinctes de X et $u \in T\left(\mathcal{F}' \cup Q, \{x_1, \ldots, x_n\}\right)$.

La relation de transition $\to_\tau$ associée au transducteur τ est définie par :

$$\begin{array}{c} \forall t, t' \in T\left(\mathcal{F} \cup \mathcal{F}' \cup Q\right), \\ t \to_\tau t' \\ \Leftrightarrow \\ \exists\, q(f\left(x_1, \ldots, x_n\right)) \to u \in \Delta, \\ \exists p \in Pos\left(t\right), \exists u_1, \ldots, u_n \in T\left(\mathcal{F}'\right), \\ t|_p = q\left(f\left(u_1, \ldots, u_n\right)\right) \\ \text{et } t' = t\left[u\left\{x_1 \leftarrow u_1, \ldots, x_n \leftarrow u_n\right\}\right]_p. \end{array}$$

Les ϵ-transitions sont un cas particulier de cette définition.

La relation de transduction induite par τ est la relation R_τ définie par : $R_\tau = \{(t, t') \mid q_0\left(t\right) \to_\tau^* t', t \in T\left(\mathcal{F}\right), t' \in T\left(\mathcal{F}'\right)\}$. Ainsi, un transducteur transforme des termes de $T\left(\mathcal{F}\right)$ en des termes de $T\left(\mathcal{F}'\right)$. On dit que le transducteur τ réalise R_τ.

Exemple 37. Considérons l'alphabet $\mathcal{F}$ à partir duquel sont définies les traces (cf. Section 2.2). Rappelons que l'alphabet $\mathcal{F}_a$ des actions se décompose en un ensemble Σ et un ensemble Γ.

Soit maintenant le transducteur $\tau = (\mathcal{F}, \mathcal{F}, Q, q_t, \Delta)$ où $Q = \{q_t, q_a, q_d\}$ et Δ est composé des règles suivantes :

- $q_t(\cdot(x, y)) \to \cdot(q_a(x), q_t(y))$;
- $q_t(\epsilon) \to \epsilon$;
- $q_a(f(x_1, \ldots, x_n)) \to f(q_d(x_1), \ldots, q_d(x_n))$ pour tout $f \in \Gamma$ d'arité $n \in \mathbb{N}$;
- $q_d(d) \to d$ pour tout $d \in \mathcal{F}_d$.

La relation de transduction induite par τ est précisément la relation identité restreinte au domaine $T_{Trace}(\mathcal{F}_\Gamma)$.

Une règle $q(f(x_1, \ldots, x_n)) \to u$ ou $q(x_1) \to u$ est *linéaire* ssi aucune variable n'apparaît plus d'une fois dans le membre droit ou dans le membre gauche. Elle est non effaçante ssi toute variable apparaissant dans le membre gauche apparaît dans le membre droit.

Un transducteur d'arbres descendant est *linéaire* (resp. non effaçant) ssi ses règles sont linéaires (resp. non effaçantes). Une relation binaire sur $T(\mathcal{F}, X) \times T(\mathcal{F}', X)$ est dite *rationnelle* ssi il existe un transducteur d'arbres descendant linéaire non effaçant qui la réalise.

La taille de τ est définie par : $|\tau| = |Q| + |\Delta|$.

L'image d'un langage régulier d'arbres de $T_{Trace}(\mathcal{F}, X)$ par une relation de transduction d'arbres rationnelle est un langage régulier d'arbres de $T_{Trace}(\mathcal{F}', X)$. Les relations de transduction d'arbres rationnelles sont fermées par union et composition fonctionnelle.

Dans la suite, l'appellation transducteur d'arbres désignera spécifiquement de tels transducteurs d'arbres descendants linéaires non effaçants.

Définition 38. Un *langage de traces* sur un alphabet $\mathcal{F}$ est un langage d'arbres dans $T_{Trace}(\mathcal{F})$. Un *automate de traces* sur $\mathcal{F}$ est un automate d'arbres reconnaissant un langage de traces. Un *transducteur de traces* est un transducteur transformant des langages de traces en des langages de traces.

Lemme 39. *Soit A un automate de traces sur un alphabet $\mathcal{F}$ et $\tau = (\mathcal{F}, \mathcal{F}', Q, q_0, \Delta)$ un transducteur de traces. Alors le langage de traces $\tau(\mathcal{L}(A))$ est reconnu par un automate de traces de taille $O(|A| \cdot |\tau|)$.*

Démonstration. Définissons deux alphabets de mots Ω et Ω' en bijection avec les ensembles finis $T_{Action}(\mathcal{F})$ et $T_{Action}(\mathcal{F}')$. Cela induit une bijection entre les mots de Ω^* (resp. Ω'^*) et les traces de $T_{Trace}(\mathcal{F})$ (resp. $T_{Trace}(\mathcal{F}')$).

Alors, A étant un automate de traces et τ un transducteur de traces, le résultat découle par analogie directe avec le cas des transducteurs de mots sur les alphabets de mots Ω et Ω' : le résultat équivalent sur les mots est montré notamment dans [87]. $\square$

5.2 Behavior Patterns

Rappelons que, dans notre formalisme, Σ désigne l'alphabet des actions concrètes composant les traces d'exécution et $\mathcal{F}_d$ désigne les données. Une action concrète est donc un terme de $T_{Action}(\mathcal{F}_\Sigma)$ de la forme $f(d_1, \ldots, d_n)$ avec $f \in \Sigma$ identifiant l'appel de librairie et $d_1, \ldots, d_n \in \mathcal{F}_d$ identifiant les arguments de l'appel. Une trace concrète est alors un terme de $T_{Trace}(\mathcal{F}_\Sigma)$ de la forme $a_1 \cdots a_n$ où $a_1, \ldots, a_n \in T_{Action}(\mathcal{F}_\Sigma)$ sont des actions concrètes. Comme dans le chapitre précédent, Γ sera l'alphabet identifiant les fonctionnalités abstraites, avec la différence qu'une action abstraite pourra avoir des paramètres de sorte *Data*.

Le problème de la détection de comportements de haut niveau peut être formalisé de la façon suivante. Premièrement, en utilisant des formules FOLTL, on définit un ensemble de behavior patterns, chaque pattern représentant un ensemble (potentiellement infini) de termes de $T_{Trace}(\mathcal{F}_\Sigma)$. Deuxièmement, on définit une relation d'abstraction R qui permet de schématiser une trace en abstrayant les occurrences des behavior patterns dans cette trace. Enfin, étant donné un programme p doté d'un ensemble infini de traces L, on formule le *problème de la détection* de la façon suivante : étant donné un comportement abstrait M défini par une formule de logique φ, existe-t-il une trace t dans $L{\downarrow_R}$ telle que $t \models \varphi$, où $L{\downarrow_R}$ est l'ensemble des formes normales de traces de L pour R ? Notre objectif est alors de trouver une méthode efficace et efficiente pour résoudre ce problème.

Comme dans le cas des mots (chapitre précédent), un behavior pattern décrit une fonctionnalité que nous souhaitons reconnaître dans une trace de programme, comme l'écriture de fichiers système, l'envoi de mails ou le ping d'hôtes distants. Une telle fonctionnalité peut être réalisée de différentes manières, selon les appels système, les appels de librairies ou le langage de programmation utilisés.

Une fonctionnalité est décrite à l'aide d'une formule FOLTL, de telle sorte que les traces validant cette formule soient les traces réalisant la fonctionnalité.

Exemple 40. Considérons la fonctionnalité d'envoi d'un ping. Une façon de réaliser cet envoi consiste en l'appel de la fonction *socket* avec le pa-

ramètre `IPPROTO_ICMP` décrivant le protocole réseau suivi d'un appel à la fonction *sendto* avec le paramètre `ICMP_ECHOREQ` décrivant les données à envoyer. Entre ces deux appels, le socket ne doit pas être fermé. Supposons qu'il existe deux constantes α et β dans $\mathcal{F}_d$ identifiant les paramètres particuliers `IPPROTO_ICMP` et `ICMP_ECHOREQ` (cf. Section 2.3 sur les traces d'exécution étendues). On décrit alors cette réalisation du ping par la formule FOLTL : $\varphi_1 = \exists x, y.\, socket\,(x, \alpha) \wedge (\neg closesocket\,(x)\ \mathbf{U}\ sendto\,(x, \beta, y))$, où le premier paramètre de *socket* est le socket créé et le second paramètre est le protocole réseau, le premier paramètre de *sendto* est le socket utilisé, le second paramètre est les données envoyées et le troisième est la cible, l'unique paramètre de *closesocket* est le socket libéré.

Un ping peut aussi être réalisé en utilisant la fonction *IcmpSendEcho*, dont le paramètre représente la cible du ping. Cela correspond à la formule FOLTL : $\varphi_2 = \exists x.\, IcmpSendEcho\,(x)$.

Ainsi, la fonctionnalité complète du ping peut être décrite par la formule FOLTL : $\varphi_{ping} = \varphi_1 \vee \varphi_2$.

On définit ensuite un behavior pattern comme l'ensemble des traces réalisant une fonctionnalité donnée, i.e. comme l'ensemble des traces validant la formule FOLTL décrivant cette fonctionnalité.

Définition 41. Un *behavior pattern* est un ensemble de traces $B \subseteq T_{Trace}\,(\mathcal{F}_\Sigma)$ validant une formule FOLTL close φ sur $AP = \Sigma$:

$$B = \{t \in T_{Trace}\,(\mathcal{F}_\Sigma) \mid t \models \varphi\}\,.$$

5.3 Problème de la Détection

La Détection

Comme dit précédemment, notre objectif est de pouvoir détecter, dans un ensemble donné de traces, un comportement prédéfini composé de combinaisons de fonctionnalités de haut niveau. Pour cela, on associe à chaque behavior pattern un symbole abstrait λ pris dans l'alphabet Γ. Un comportement abstrait décrit alors des combinaisons des symboles abstraits associés aux behavior patterns et est défini à l'aide d'une formule FOLTL sur $AP = \Gamma$ (au lieu de Σ, comme c'était le cas pour les behavior patterns).

Définition 42. Un *comportement abstrait* est un ensemble de traces $M \subseteq T_{Trace}\,(\mathcal{F}_\Gamma)$ validant une formule FOLTL close φ_M sur $AP = \Gamma$:

$$M = \{t \in T_{Trace}\,(\mathcal{F}_\Gamma) \mid t \models \varphi_M\}\,.$$

Lorsque M est défini par une formule φ_M, on écrit : $M := \varphi_M$.

Exemple 43. Le comportement abstrait du ping d'un hôte distant peut alors être défini par la formule : $\varphi_M = \exists x. \mathbf{F}\, \lambda_{ping}(x)$.

Dans la suite, par souci de clarté, l'opérateur initial $\mathbf{F}$ sera implicite dans les définitions des comportements abstraits.

Supposons maintenant que nous souhaitions détecter un comportement abstrait dans l'ensemble de traces L d'un certain programme. Pour comparer ces traces au comportement abstrait, il faut considérer les occurrences des behavior patterns qu'elles contiennent, au niveau abstrait. Pour cela, on définit une *relation d'abstraction* R, qui marque de telles occurrences dans les traces en insérant un symbole abstrait λ_B lorsqu'une occurrence du behavior pattern B est identifiée.

Désormais, si un behavior pattern est défini à l'aide d'une formule FOLTL φ et est associé à un symbole d'abstraction λ, on pourra le décrire par la notation $\lambda := \varphi$.

Par ailleurs, le symbole d'abstraction peut avoir des paramètres correspondant à ceux utilisés par le behavior pattern. Cela permet d'exprimer dans un comportement abstrait des contraintes sur le flux de données. Par exemple, le symbole d'abstraction du behavior pattern du ping pourrait prendre un paramètre dénotant la cible du ping. La signature d'un déni de service pourrait alors être définie, par exemple, comme une séquence de 100 pings ayant la même cible.

Exemple 44. Le behavior pattern du ping, défini dans l'Exemple 40, est abstrait dans les traces en insérant le symbole λ_{ping} après l'action *send* ou après l'action *IcmpSendEcho*. La trace $socket(1, \alpha) \cdot gethostbyname(2) \cdot sendto(1, \beta, 3) \cdot closesocket(1)$ peut alors être abstraite en la trace $socket(1, \alpha) \cdot gethostbyname(2) \cdot sendto(1, \beta, 3) \cdot \lambda_{ping}(3) \cdot closesocket(1)$.

Une relation d'abstraction est donc définie comme une relation binaire de $T_{Trace}(\mathcal{F}) \times T_{Trace}(\mathcal{F})$ transformant une trace en une autre trace en insérant une action abstraite de la forme $\lambda_B(d_1, \ldots, d_n)$ lorsqu'un behavior pattern B est identifié, avec $d_1, \ldots, d_n \in \mathcal{F}_d$. Ainsi, l'abstraction d'une trace révèle des combinaisons de behavior patterns abstraits, qui peuvent constituer le comportement abstrait à observer. Dans la Section 5.5, on définit formellement la relation d'abstraction comme une relation de réduction induite par un système de réécriture de termes. En outre, cette relation est terminante car une occurrence d'un behavior pattern ne pourra être abstraite qu'une seule fois et

l'action abstraite insérée n'introduit pas de nouvelles opportunités d'abstraction : toute chaîne d'abstractions est donc finie et toute trace admet au moins une forme normale par rapport à R.

Rappelons que $L{\downarrow_R}$ est l'ensemble des formes normales des traces de L pour R. Le problème de la détection peut alors être formulé comme suit.

Définition 45. Soit M un comportement abstrait défini par une formule φ_M. Un ensemble de traces L *exhibe le comportement* M, noté $L \Cap M$, ssi :

$$\exists t \in \ L{\downarrow_R}|_\Gamma \, , \, t \models \varphi_M.$$

Remarque 46. Nous distinguons la notion d'exhibition d'un comportement abstrait de la notion de réalisation. On dira qu'une trace t réalise un comportement abstrait M ssi $t \in M$ tandis qu'elle exhibe un comportement abstrait M ssi une de ses formes complètement abstraites réalise M.

La (m, n)-complétude

Lorsque L est restreint à une seule trace ou à un ensemble fini de traces, par exemple en analyse dynamique, $L{\downarrow_R}$ est calculable dès lors que la relation R est terminante, ce qui est le cas. De plus, la quantification FOLTL étant réalisée sur des variables de domaine fini $\mathcal{F}_d$, la validation de formules FOLTL est décidable, donc on peut également décider si L exhibe M.

Lorsque L est un ensemble infini de traces, le calcul de $L{\downarrow_R}$ repose en général sur le calcul de l'ensemble $R^*(L)$ des descendants de L par R. Mais $R^*(L)$ n'est calculable que pour certaines classes de systèmes de réécriture [47, 48] et lorsque L est régulier. Malheureusement, les systèmes de réécriture réalisant les relations d'abstraction et décrits en Section 5.5 n'appartiennent à aucune de ces classes (nous avons par exemple essayé de simuler l'abstraction à l'aide d'un système linéaire semi-monadique, d'un système décroissant, d'un système linéaire généralisé semi-monadique, d'un système linéaire droit à recouvrement sur chemins finis, tous décrits dans [47], sans succès). Il en résulte que nous ne pouvons pas décider si L exhibe M en passant par une construction de $L{\downarrow_R}$.

Néanmoins, nous allons voir que, pour les comportements considérés en pratique, une abstraction partielle de l'ensemble de traces suffit, autrement dit qu'il n'est pas nécessaire de calculer l'ensemble des formes normales. On propose donc un algorithme de détection reposant sur une approximation sûre de l'ensemble des traces abstraites. Cette approximation doit être choisie avec précaution. Par exemple, elle ne peut consister en un calcul, pour un certain n, de l'ensemble $R^{\leq n}(L)$ des descendants de L jusqu'à l'ordre n, comme le montre l'exemple suivant.

Exemple 47. Soit $\lambda_1 := a$, $\lambda_2 := b$, $\lambda_3 := c$ trois behavior patterns associés à des relations d'abstraction insérant le symbole d'abstraction après a, b et c respectivement. Soit $M := \lambda_1 \wedge (\neg\lambda_2 \,\mathbf{U}\, \lambda_3)$ un comportement abstrait. Supposons qu'il existe une borne n telle que $L{\downarrow_R}$ puisse être approximée par $R^{\leq n}(L)$ dans la Définition 45. La trace $t = a^{n-1} \cdot b \cdot c \cdot d$ est un exemple d'une trace saine. Pourtant la trace $t' = (a \cdot \lambda_1)^{n-1} \cdot b \cdot c \cdot \lambda_3 \cdot d$ est dans $R^{\leq n}(\{t\})$ et sa projection sur Γ est dans M, donc on déduirait à tort que t exhibe M.

Le problème vient de ce que $R^{\leq n}(L)$ peut contenir des traces partiellement abstraites qui paraissent exhiber le comportement abstrait bien que quelques étapes d'abstraction supplémentaires leur aurait fait quitter la signature.

On veut donc exclure de $R^{\leq n}(L)$ les traces ne réalisant pas le comportement abstrait de façon fiable, sans toutefois avoir à atteindre les formes normales. En fait, on identifie une propriété fondamentale que nous appelons (m, n)-complétude, aisément vérifiée par les comportements abstraits en pratique dans le domaine de la détection de codes malicieux. Cette propriété établit que, pour qu'un programme exhibe un comportement abstrait, une condition nécessaire et suffisante est la suivante : il existe une trace partiellement abstraite, obtenue en au plus m étapes d'abstraction, qui réalise le comportement et dont les descendants jusqu'à l'ordre n le réalisent toujours.

Définition 48. Soit M un comportement abstrait défini par une formule φ_M et m et n des entiers positifs. M a la propriété de (m, n)-*complétude* ssi pour tout ensemble de traces $L \subseteq T_{Trace}(\mathcal{F}_\Sigma)$:

$$\begin{gathered} L \Cap M \\ \Leftrightarrow \\ \exists t' \in R^{\leq m}(L),\, \forall t'' \in R^{\leq n}(t')\big|_\Gamma,\, t'' \models \varphi_M. \end{gathered}$$

On montre ensuite, dans la section suivante, que lorsque L est régulier, il existe une procédure de détection correcte et complète pour tout comportement abstrait ayant cette propriété. De plus, la complexité en temps et espace de cette procédure est linéaire en la taille de la représentation de L.

Les théorèmes suivants montrent que la propriété de (m, n)-complétude est réaliste pour les comportements abstraits considérés en pratique.

Pour cela, on montre tout d'abord, dans le Lemme 51 suivant, que, dès lors qu'un behavior pattern est abstrait dans une trace t après un nombre quelconque d'étapes d'abstraction, il peut être abstrait dans t à la même position concrète et dès la première étape d'abstraction.

Définition 49 (Position concrète)**.** Soit t un terme de $T_{Trace}(\mathcal{F})$ et t' un sous-terme de t, de sorte *Trace*. La position *concrète de* t' dans t est la position de $t'|_\Sigma$ dans $t|_\Sigma$.

Définition 50 (Abstraction à une position concrète)**.** Soit B un behavior pattern associé à un symbole d'abstraction $\lambda \in \Gamma$ et à une relation d'abstraction $\rightarrow$. On dit que la trace $t = t_1 \cdot t_2$ est *abstraite par rapport à* B *en* $t_1 \cdot \lambda \cdot t_2$ *à la position concrète* p, noté $t_1 \cdot t_2 \rightarrow_p t_1 \cdot \lambda \cdot t_2$, ssi $t_1 \cdot t_2 \rightarrow t_1 \cdot \lambda \cdot t_2$ et p est la position concrète de t_2 dans t.

Avec ces définitions, le lemme à démontrer peut être exprimé de la façon suivante. Si $t \rightarrow^* t_1 \cdot t_2 \rightarrow_p t_1 \cdot \lambda \cdot t_2$, alors il existe $u_1, u_2 \in T(\mathcal{F})$ tels que : $t \rightarrow_p u_1 \cdot \lambda \cdot u_2$.

$$\begin{tikzcd}
\mathrm{t} \arrow[r, "*"] \arrow[d, "\mathrm{p}"] & t_1 \cdot t_2 \arrow[d, "\mathrm{p}"] \\
u_1 \cdot \lambda \cdot u_2 & t_1 \cdot \lambda \cdot t_2
\end{tikzcd}$$

On montre en fait une forme plus générale de ce lemme, où un nombre variable de behavior patterns (pas nécessairement distincts) sont abstraits l'un après l'autre.

Lemme 51. *Soit une trace* $t \in T_{Trace}(\mathcal{F})$ *et des actions abstraites* α_1, α_2, $\ldots$ $\alpha_k \in T_{Action}(\mathcal{F}_\Gamma)$. *Soit la chaîne d'abstractions à partir de* t *suivante :* $t \rightarrow^* t_1 \cdot t'_1 \rightarrow_{p_1} t_1 \cdot \alpha_1 \cdot t'_1 \rightarrow^* t_2 \cdot t'_2 \rightarrow_{p_2} t_2 \cdot \alpha_2 \cdot t'_2 \rightarrow^* \ldots \rightarrow^* t_k \cdot t'_k \rightarrow_{p_k} t_k \cdot \alpha_k \cdot t'_k$ *où on distingue* k *étapes d'abstraction. Alors :*

$$\exists u_1, \ldots u_k, u'_1, \ldots u'_k \in T_{Trace}(\mathcal{F}),$$
$$t \rightarrow_{p_1} u_1 \cdot \alpha_1 \cdot u'_1 \rightarrow_{p_2} u_2 \cdot \alpha_2 \cdot u'_2 \rightarrow_{p_3} \ldots \rightarrow_{p_k} u_k \cdot \alpha_k \cdot u'_k.$$

Démonstration. Par induction sur la longueur l de la dérivation $t \rightarrow^* t_k \cdot \alpha_k \cdot t'_k$.

- Pour le cas $l = 1$, on a : $t \rightarrow_{p_1} t_1 \cdot \alpha_1 \cdot t'_1$. Il existe donc u_1, u'_1, $u_1 = t_1, u'_1 = t'_1$.
- Pour l'étape générale d'induction, supposons que nous ayons cette propriété pour $l = n$. On prouve la propriété pour $l = n + 1$. Par hypothèse d'induction appliquée à $t \rightarrow^* t_1 \cdot t'_1 \rightarrow_{p_1} t_1 \cdot \alpha_1 \cdot t'_1 \rightarrow^* t_2 \cdot t'_2 \ldots \rightarrow^* t_k \cdot t'_k \rightarrow_{p_k} t_k \cdot \alpha_k \cdot t'_k$, on déduit :

  $$\exists u_1, \ldots u_k, u'_1, \ldots, u'_k \in T_{Trace}(\mathcal{F}), t \rightarrow_{p_1} u_1 \cdot \alpha_1 \cdot u'_1 \rightarrow \ldots \rightarrow u_k \cdot \alpha_k \cdot u'_k.$$

Pour $l = n+1$, la chaîne de longueur n est étendue par $t_k \cdot \alpha_k \cdot t'_k \to t_{k+1} \cdot \alpha_{k+1} \cdot t'_{k+1}$.
On veut réécrire $u_k \cdot \alpha_k \cdot u'_k$ en $u_{k+1} \cdot \alpha_{k+1} \cdot u'_{k+1}$.
Or, l'existence de la réduction $t_k \cdot \alpha_k \cdot t'_k \to t_{k+1} \cdot \alpha_{k+1} \cdot t'_{k+1}$ entraîne l'existence d'une occurrence du behavior pattern B_{k+1} dans $t_k \cdot \alpha_k \cdot t'_k$. Cette occurrence apparaît aussi dans $u_k \cdot \alpha_k \cdot u'_k$ et peut donc être abstraite à la même position concrète p_{k+1}, d'où l'existence de termes u_{k+1} et u'_{k+1} tels que : $u_k \cdot \alpha_k \cdot u'_k \to_{p_{k+1}} u_{k+1} \cdot \alpha_{k+1} \cdot u'_{k+1}$.

□

On prouve alors que les comportements abstraits simples, décrivant des séquences d'actions abstraites sans contraintes autre que des contraintes de flux de données, ont la propriété de (m, n)-complétude.

Théorème 52. *Soit Y un ensemble de variables de sorte Data. Soit $\alpha_1, \ldots, \alpha_m \in T_{Action}(\mathcal{F}_\Gamma, Y)$. Alors le comportement abstrait $M := \exists Y.\, \alpha_1 \odot \alpha_2 \odot \ldots \odot \alpha_m$ a la propriété de $(m, 0)$-complétude.*

Démonstration. Soit $\varphi_M = \exists Y.\, \mathbf{F}(\alpha_1 \odot \alpha_2 \odot \ldots \odot \alpha_m)$.

Soit $L \subseteq T_{Trace}(\mathcal{F}_\Sigma)$ un ensemble de traces. On montre que :

$$L \Cap M$$
$$\Leftrightarrow$$
$$\exists t' \in R^{\leq m}(L), \forall t'' \in R^{\leq 0}(t')\big|_\Gamma, t'' \models \varphi_M.$$

D'abord, par la sémantique de FOLTL, une trace $t \in T_{Trace}(\mathcal{F}_\Gamma)$ valide la formule $\exists Y.\, \mathbf{F}(\alpha_1 \odot \alpha_2 \odot \ldots \odot \alpha_m)$ ssi t s'écrit $t = t'_1 \cdot t'_2$ avec $t'_1, t'_2 \in T_{Trace}(\mathcal{F}_\Gamma)$ et t'_2 validant la formule $\exists Y.\, \alpha_1 \odot \alpha_2 \odot \ldots \odot \alpha_m$. Puis t'_2 valide la formule $\exists Y.\, \alpha_1 \odot \alpha_2 \odot \ldots \odot \alpha_m$ ssi il existe une instanciation $\sigma_Y \in Inst_Y$ telle que t'_2 valide la formule $\alpha_1\sigma_Y \odot \alpha_2\sigma_Y \odot \ldots \odot \alpha_m\sigma_Y$, équivalente à la formule $\alpha_1\sigma_Y \wedge \mathbf{X}(\top\, \mathbf{U}\, \alpha_2\sigma_Y \wedge \mathbf{X}(\top\, \mathbf{U}\, \ldots \wedge \mathbf{X}(\top\, \mathbf{U}\, \alpha_m\sigma_Y)))$. Donc la trace t'_2 valide les formules $\alpha_1\sigma_Y$ et $\mathbf{X}(\top\, \mathbf{U}\, \alpha_2\sigma_Y \wedge \mathbf{X}(\top\, \mathbf{U}\, \ldots \wedge \mathbf{X}(\top\, \mathbf{U}\, \alpha_m\sigma_Y)))$. La trace t'_2 est donc de la forme :

$$t'_2 = \alpha_1\sigma_Y \cdot t_1 \cdot \alpha_2\sigma_Y \cdot t_2 \cdots \alpha_m\sigma_Y \cdot t_m$$

où $t_1, \ldots, t_m \in T_{Trace}(\mathcal{F}_\Gamma)$.

$\Rightarrow$: Par la Définition 45, il existe une trace $t \in L$ avec une forme normale $t{\downarrow}$ telle que $t{\downarrow}|_\Gamma$ valide φ_M. $t{\downarrow}$ s'écrit donc :

$$t{\downarrow} = t_0 \cdot \alpha_1\sigma_Y \cdot t_1 \cdot \alpha_2\sigma_Y \cdot t_2 \cdots \alpha_m\sigma_Y \cdot t_m$$

où $t_0, \ldots, t_m \in T_{Trace}(\mathcal{F})$.

Par le Lemme 51 appliqué à la dérivation de t vers $t{\downarrow}$, il existe donc $u_0, \ldots, u_m$ dans $T_{Trace}(\mathcal{F})$ tels que t s'abstrait en $t' = u_0 \cdot \alpha_1 \sigma_Y \cdot u_1 \cdots \alpha_m \sigma_Y \cdot u_m$ en exactement m étapes. Donc $t' \in R^{\leq m}(L)$. De plus, $R^{\leq 0}(t')\big|_\Gamma = \{t'|_\Gamma\}$ et $t'|_\Gamma \models \varphi_M$.

$\Leftarrow$: Soit $t' \in R^{\leq m}(L)$ une forme partiellement abstraite d'une trace de L telle que $\forall t'' \in R^{\leq 0}(t')\big|_\Gamma, t'' \models \varphi_M$. Alors t' s'écrit $t' = t_0 \cdot \alpha_1 \sigma_Y \cdot t_1 \cdots \alpha_m \sigma_Y \cdot t_m$, où $t_0, \ldots, t_m \in T_{Trace}(\mathcal{F})$ et $\sigma_Y \in Inst_Y$. Clairement, toute abstraction future de t' sera toujours de la forme $u_0 \cdot \alpha_1 \sigma_Y \cdot u_1 \cdots \alpha_m \sigma_Y \cdot u_m$ et ce sera en particulier vrai de toute forme normale $t'{\downarrow}$ de t' par R. Donc $t'{\downarrow}\big|_\Gamma \models \varphi_M$ et $L \Cap M$. □

Un comportement intéressant en pratique

On montre désormais que des comportement abstraits plus complexes, interdisant des actions abstraites spécifiques, ont également la propriété de (m, n)-complétude.

Pour un behavior pattern λ, notons R_λ la restriction de la relation d'abstraction R à l'abstraction par rapport à λ. On dit que deux behavior patterns λ et λ' sont *indépendants* ssi : $R_\lambda \circ R_{\lambda'} = R_{\lambda'} \circ R_\lambda$. Dans la pratique, l'indépendance des behavior patterns est facile à assurer, comme nous le verrons dans la Section 5.7.

On a alors le résultat suivant.

Théorème 53. *Soit* $M := \exists Y.\, \lambda_1(\overline{x_1}) \wedge \neg(\exists Z.\, \lambda_2(\overline{x_2}))\, \mathbf{U}\, \lambda_3(\overline{x_3})$ *un comportement abstrait où* Y *et* Z *sont des ensembles disjoints de variables de sorte Data et où* $\lambda_2 \neq \lambda_1$, $\lambda_2 \neq \lambda_3$ *et* λ_2 *est indépendant de* λ_3.

Alors M *a la propriété de* $(2, 1)$*-complétude.*

Démonstration. Soit $\varphi_M = \exists Y.\, \mathbf{F}(\lambda_1(\overline{x_1}) \wedge \neg(\exists Z.\, \lambda_2(\overline{x_2}))\, \mathbf{U}\, \lambda_3(\overline{x_3}))$.

Notons α_1, α_2 et α_3 les actions $\lambda_1(\overline{x_1})$, $\lambda_2(\overline{x_2})$ et $\lambda_3(\overline{x_3})$, respectivement.

Soit $L \subseteq T_{Trace}(\mathcal{F}_\Sigma)$ un ensemble de traces. On montre que :

$$L \Cap M$$
$$\Leftrightarrow$$
$$\exists t' \in R^{\leq 2}(L), \forall t'' \in R^{\leq 1}(t')\big|_\Gamma, t'' \models \varphi_M.$$

$\Rightarrow$ Par définition de l'infection, il existe une trace $t \in L$ telle qu'une de ses formes normales $t{\downarrow}$ valide φ_M et s'écrive donc :

$$t{\downarrow} = t_1 \cdot \alpha_1 \sigma_Y \cdot t_2 \cdot \alpha_3 \sigma_Y \cdot t_3$$

où $\sigma_Y \in Inst_Y$ est une instanciation sur Y, $t_1, t_2, t_3 \in T_{Trace}(\mathcal{F})$ et il n'existe pas d'instanciation σ_Z telle que $\alpha_2\sigma_Y\sigma_Z$ apparaisse dans t_2.

On définit tout d'abord un terme $t' \in R^{\leq 2}(t)$ qui contient la même occurrence $\alpha_1\sigma_Y \cdot \alpha_3\sigma_Y$ de M et on montre que ses abstractions futures réalisent toujours M. En effet, étant donné que $t{\downarrow} = t_1 \cdot \alpha_1\sigma_Y \cdot t_2 \cdot \alpha_3\sigma_Y \cdot t_3$, il existe, par le Lemme 51, des traces u', v', w' de $T_{Trace}(\mathcal{F}_\Sigma)$ telles que :

$$t \to\to u' \cdot \alpha_1\sigma_Y \cdot v' \cdot \alpha_3\sigma_Y \cdot w'.$$

On définit donc $t' = u' \cdot \alpha_1\sigma_Y \cdot v' \cdot \alpha_3\sigma_Y \cdot w'$. De plus, la trace $t \in L$ étant concrète, v' ne contient pas d'action abstraite, donc elle ne contient aucune instance de $\alpha_2\sigma_Y$. On en déduit : $t'|_\Gamma \models \varphi_M$.

On montre maintenant que : $\forall t'' \in R^{\leq 1}(t')|_\Gamma, t'' \models \varphi_M$. En fait, on montre plus généralement que : $\forall t'' \in R^*(t')|_\Gamma, t'' \models \varphi_M$. Supposons que ce ne soit pas le cas et que t' puisse donc être réécrit de telle sorte qu'une action $\alpha_2\sigma_Y\sigma'_Z$ soit insérée dans v' pour une certaine instanciation $\sigma'_Z \in Inst_Z$. L'occurrence du behavior pattern lié à cette insertion doit aussi apparaître dans $t{\downarrow}$. Or $t{\downarrow}$ est en forme normale donc cette occurrence a déjà été abstraite, à la même position concrète, dans un terme de la chaîne d'abstractions de t à $t{\downarrow}$, c'est-à-dire après une action concrète de t_2.

De plus, par hypothèse, dans $t{\downarrow}$, aucune instance de $\alpha_2\sigma_X$ n'apparaît dans t_2 puisque $t{\downarrow}$ valide φ_M.

Donc une action $\alpha_2\sigma_Y\sigma'_Z$ doit nécessairement apparaître dans les actions abstraites en tête de t_3. Mais, λ_2 et λ_3 étant indépendants, c'est impossible. Ainsi, aucune abstraction de t' n'a pu insérer cette action $\alpha_2\sigma_Y\sigma'_Z$ entre $\alpha_1\sigma_Y$ et $\alpha_3\sigma_Y$.

On en déduit : $\forall t'' \in R^*(t')|_\Gamma, t'' \models \varphi_M$.

$\Leftarrow$ On raisonne par contradiction. Soit $t' \in R^{\leq 2}(L)$ une trace telle que $\forall t'' \in R^{\leq 1}(t')|_\Gamma, t'' \models \varphi_M$. En supposant que L n'exhibe pas M, on construit une trace $t'_1 \in R^{\leq 1}(t')$ qui ne réalise pas M et on utilise l'hypothèse $\forall t'' \in R^{\leq 1}(t')|_\Gamma, t'' \models \varphi_M$ pour obtenir une contradiction.

En particulier, $t'|_\Gamma \models \varphi_M$ donc, comme dans la preuve du Théorème 52, par définition de la satisfiabilité de la formule $\varphi_M = \exists Y. \mathbf{F}(\lambda_1(\overline{x_1}) \wedge \neg(\exists Z. \lambda_2(\overline{x_2}))\ \mathbf{U}\ \lambda_3(\overline{x_3}))$, il existe une instanciation $\sigma_Y \in Inst_Y$ telle qu'on puisse décomposer t' en :

$$t' = t_1 \cdot \alpha_1\sigma_Y \cdot t_2 \cdot \alpha_3\sigma_Y \cdot t_3$$

où $t_1, t_2, t_3 \in T_{Trace}(\mathcal{F})$, et tel qu'il n'existe aucune instanciation $\sigma_Z \in Inst_Z$ telle que $\alpha_2\sigma_Y\sigma_Z$ apparaisse dans t_2.

Supposons que L n'exhibe pas M. Soit t'' une forme normale de t' : $t'' \in \{t'\}\big\downarrow_R$. Alors, par la Définition 45, $t''|_\Gamma \not\models \varphi_M$. Par définition de la satisfiabilité de la formule $\varphi_M = \exists Y.\, \mathbf{F}(\lambda_1\,(\overline{x_1}) \wedge \neg\,(\exists Z.\, \lambda_2\,(\overline{x_2}))\, \mathbf{U}\, \lambda_3\,(\overline{x_3}))$, il doit exister une instanciation $\sigma_Z \in Inst_Z$ telle qu'une action abstraite $\alpha_2\sigma_Y\sigma_Z$ ait été insérée dans un terme de la dérivation de t' à t'', à une position concrète p entre $\alpha_1\sigma_Y$ et $\alpha_3\sigma_Y$. Par le Lemme 51, on aurait pu insérer cette action $\alpha_2\sigma_Y\sigma_Z$ directement dans le terme t', à la même position concrète p :

$$\exists u, w,\, t' \to_p u \cdot \alpha_2\sigma_Y\sigma_Z \cdot w.$$

L'insertion ayant eu lieu entre les actions $\alpha_1\sigma_Y$ et $\alpha_3\sigma_Y$, et étant donné que $t' = t_1 \cdot \alpha_1\sigma_Y \cdot t_2 \cdot \alpha_3\sigma_Y \cdot t_3$, on peut décomposer t_2 en $t_2 = t_2^1 \cdot t_2^2$ de telle sorte que l'insertion ait lieu après t_2^1, autrement dit :

$$t' \to_p t_1 \cdot \alpha_1\sigma_Y \cdot t_2^1 \cdot \alpha_2\sigma_Y\sigma_Z \cdot t_2^2 \cdot \alpha_3\sigma_Y \cdot t_3.$$

Notons t'_1 le terme obtenu : $t'_1 = t_1 \cdot \alpha_1\sigma_Y \cdot t_2^1 \cdot \alpha_2\sigma_Y\sigma_Z \cdot t_2^2 \cdot \alpha_3\sigma_Y \cdot t_3$. Alors $t'_1 \in R^{\leq 1}\,(t')$ et pourtant $t'_1|_\Gamma \not\models \varphi_M$, ce qui contredit l'hypothèse $\forall t'' \in R^{\leq 1}\,(t')\big|_\Gamma,\, t'' \models \varphi_M$. □

En pratique, comme nous le verrons en Section 5.6, la plupart des signatures sont des disjonctions de formules de la forme : $\exists Y.\, \alpha_1 \odot \alpha_2 \odot \ldots \odot \alpha_m$, comme dans le Théorème 52, ou de la forme :

$$\begin{aligned} \exists Y.\lambda_1\,(\overline{x_1}) \wedge \quad & \neg\,(\exists Z_1.\, \lambda\,(\overline{z_1}))\ \mathbf{U}\, \lambda_2\,(\overline{x_2}) \wedge \\ & \neg\,(\exists Z_2.\, \lambda\,(\overline{z_2}))\ \mathbf{U}\ \ldots \lambda_k\,(\overline{x_k})\,. \end{aligned}$$

où λ est indépendant de $\lambda_2, \ldots, \lambda_k$. D'après la preuve du Théorème 53, on conjecture que la seconde formule a la propriété de $(k, 1)$-complétude.

La condition d'indépendance n'est pas nécessaire en général, pour garantir que de tels comportement abstraits ont une propriété de (m, n)-complétude pour des entiers m et n, mais l'absence de cette condition résulte en des valeurs de m et n sensiblement plus élevées.

Fondamentalement, d'après la Définition 45, la détection d'un comportement abstrait se décompose en deux étapes indépendantes : une étape d'abstraction (calcul de $L\downarrow_R$) suivie d'une étape de vérification. La première étape calcule les formes abstraites des traces du programme tandis que la deuxième étape applique les techniques de vérification usuelles afin de décider si l'une des traces abstraites calculées valide la formule FOLTL définissant le comportement abstrait. Cependant, lorsque l'on utilise la propriété de (m, n)-complétude pour contourner l'indécidabilité générale du calcul de $L\downarrow_R$ lors

de l'étape d'abstraction et lorsque plus particulièrement $n > 0$, comme dans le Théorème 53, on fusionne ces deux étapes en construisant d'abord l'ensemble $\left\{t \in T_{Trace}(\mathcal{F}),\, R^{\leq n}(t) \models \varphi_M\right\}$ puis en l'intersectant avec $R^{\leq m}(L)$. L'algorithme de détection ne reposant plus sur la validation d'une formule FOLTL, on n'exploite alors pas les résultats puissants de la théorie du model checking.

Dans le cas de $M := \exists Y.\, \lambda_1(\overline{x_1}) \wedge \neg(\exists Z.\, \lambda_2(\overline{x_2}))\, \mathbf{U}\, \lambda_3(\overline{x_3})$, on montre donc que la (m,n)-complétude du comportement M permet de déduire un algorithme de détection qui préserve la décomposition précédente et qui est décidable dès lors que les relations R et $R_{\lambda_2}\!\downarrow$ sont rationnelles, ce qui est le cas en pratique.

Théorème 54. *Soit M un comportement abstrait défini par une formule $\varphi_M = \exists Y.\, \lambda_1(\overline{x_1}) \wedge \neg(\exists Z.\, \lambda_2(\overline{x_2}))\, \mathbf{U}\, \lambda_3(\overline{x_3})$ où Y et Z sont des ensembles disjoints de variables de sorte Data et où $\lambda_2 \neq \lambda_1$, $\lambda_2 \neq \lambda_3$ et λ_2 est indépendant de λ_3.*

Alors, pour tout ensemble de traces $L \subseteq T_{Trace}(\mathcal{F}_\Sigma)$, L exhibe M ssi :

$$\exists t \in R_{\lambda_2}\!\downarrow (R^{\leq 2}(L))\big|_\Gamma,\, t \models \varphi_M.$$

Démonstration. $\Rightarrow$: Par le Théorème 53, M a la propriété de $(2,1)$-complétude donc, par la Définition 48, il existe une trace $t' \in R^{\leq 2}(L)$ telle que $\forall t'' \in R^{\leq 1}(t')\big|_\Gamma,\, t'' \models \varphi_M$. En fait, on a montré dans la preuve du Théorème 53 que : $\forall t'' \in R^*(t')|_\Gamma,\, t'' \models \varphi_M$. Étant donné que $R_{\lambda_2}\!\downarrow \subseteq R^*_{\lambda_2} \subseteq R^*$, on a :

$$\forall t'' \in R_{\lambda_2}\!\downarrow (t'),\, t''\big|_\Gamma \models \varphi_M.$$

La relation d'abstraction R étant terminante, l'ensemble $R_{\lambda_2}\!\downarrow (t')$ n'est pas vide, ce qui entraîne :

$$\exists t'' \in R_{\lambda_2}\!\downarrow (t'),\, t''\big|_\Gamma \models \varphi_M.$$

En observant que, par hypothèse, $t' \in R^{\leq 2}(L)$, on obtient :

$$\exists t \in R_{\lambda_2}\!\downarrow \left(R^{\leq 2}(L)\right)\big|_\Gamma,\, t \models \varphi_M.$$

$\Leftarrow$: Soit $t'' \in R_{\lambda_2}\!\downarrow \left(R^{\leq 2}(L)\right)$ une trace telle que : $t''|_\Gamma \models \varphi_M$. Toute occurrence du behavior pattern λ_2 a été abstraite dans t'' donc toute abstraction future de t'' par R n'insère que des actions abstraites de $T_{Action}\left(\mathcal{F}_{\Gamma\setminus\{\lambda_2\}}\right)$, d'où : $\forall u \in R^*(t'')|_\Gamma,\, u \models \varphi_M$. Donc toute forme normale de t'' par R réalise M. La relation R étant terminante, t'' a au moins une forme normale par rapport à R donc, par la Définition 45, L exhibe M. $\square$

Lorsqu'à la fois la relation d'abstraction R et la relation $R_{\lambda_2}\downarrow$ sont rationnelles, l'ensemble $R_{\lambda_2}\downarrow(R^{\leq 2}(L))$ est calculable et régulier, et la détection se ramène alors à un problème classique de model checking. Dans le cas général, $R_{\lambda_2}\downarrow$ n'est pas *rationnelle* mais, dans nos expérimentations, la forme particulière du behavior pattern λ_2 rendra cette relation rationnelle, i.e. on pourra inférer de la définition du behavior pattern λ_2 un transducteur réalisant $R_{\lambda_2}\downarrow$.

Remarque 55. Une définition équivalente de l'infection pourrait consister à compiler le comportement abstrait, c'est-à-dire à calculer l'ensemble $\pi_\Gamma^{-1}(M)\big\downarrow_{R^{-1}}$ des traces concrètes exhibant M. Un ensemble de traces L exhiberait alors M ssi l'une de ses traces était dans cet ensemble. Cette définition paraît plus intuitive : plutôt que d'abstraire une trace et de la comparer à un comportement abstrait, on vérifie si cette trace est une implantation du comportement. Cependant, cette approche requiert de d'abord calculer la forme compilée du comportement abstrait, $\pi_\Gamma^{-1}(M)\big\downarrow_{R^{-1}}$, qui n'est généralement pas calculable et dont la représentation, lorsqu'elle est calculable, peut avoir une complexité prohibitive provenant de l'entrelacement des occurrences des behavior patterns (en particulier lorsque les traces réalisant ces behavior patterns sont complexes) et provenant de l'instanciation des variables.

5.4 Complexité de la Détection

Le problème de la détection, comme le problème plus général de l'analyse de programmes, requiert de calculer une abstraction partielle de l'ensemble des traces possibles. En pratique, pour manipuler cet ensemble, on considère une approximation régulière de cet ensemble, autrement dit on considère un automate d'arbres le représentant. De plus, nous verrons en Section 5.5 qu'en pratique la relation d'abstraction est rationnelle, autrement dit elle est réalisée par un transducteur d'arbres. Cette rationalité garantit d'une part la décidabilité du calcul de l'abstraction partielle d'un automate de traces et donc de la détection, et d'autre part l'efficacité des algorithmes associés.

Pour commencer, on définit l'ensemble des traces réalisant M et dont les descendants jusqu'à l'ordre n réalisent toujours M.

Définition 56. Soit R une relation d'abstraction. Soit M un comportement abstrait défini par une formule φ_M ayant la propriété de (m,n)-complétude. L'*ensemble des traces n-exhibant M* par rapport à R est l'ensemble :

$$\{t \in T_{Trace}(\mathcal{F}) \mid \forall t' \in R^{\leq n}(t),\ t'\big|_\Gamma \models \varphi_M\}.$$

On montre alors que cet ensemble est régulier.

Lemme 57. *Soit R une relation d'abstraction. Soit M un comportement abstrait ayant la propriété de (m,n)-complétude pour des entiers positifs m et n.*

Si R^{-1} est rationnelle, alors l'ensemble des traces n-exhibant M par rapport à R est régulier.

Démonstration. Soit π_Γ la projection sur Γ. On note $\pi_\Gamma^{-1} \subseteq T_{Trace}(\mathcal{F}_\Gamma) \times T_{Trace}(\mathcal{F})$ sa relation inverse.

Définissons $M' = \pi_\Gamma^{-1}(M)$ et notons M'' l'ensemble des traces n-exhibant M par rapport à R. On a alors :

$$\begin{aligned} M'' = & \{t \in T_{Trace}(\mathcal{F}) \mid \forall t' \in R^{\leq n}(t),\ t'|_\Gamma \models \varphi_M\} \\ = & \{t \in T_{Trace}(\mathcal{F}) \mid \forall t' \in R^{\leq n}(t),\ t'|_\Gamma \in M\} \\ = & \{t \in T_{Trace}(\mathcal{F}) \mid \forall t' \in R^{\leq n}(t),\ t' \in \pi_\Gamma^{-1}(M)\} \\ = & \{t \in T_{Trace}(\mathcal{F}) \mid \forall t' \in R^{\leq n}(t),\ t' \in M'\}. \end{aligned}$$

Observons maintenant que, pour tout ensemble $A \subseteq T_{Trace}(\mathcal{F})$, tout terme $t \in T_{Trace}(\mathcal{F})$ et tout entier $i \in \mathbb{N}$, t peut être réécrit par R en un terme de A en i étapes ssi un terme de A peut être réécrit par R^{-1} en t en i étapes :

$$R^i(t) \cap A \neq \emptyset \Leftrightarrow t \in \left(R^{-1}\right)^i(A). \tag{5.1}$$

On en déduit :

$$\begin{gathered} t \in M'' \\ \Leftrightarrow \\ R^{\leq n}(t) \subseteq M' \\ \Leftrightarrow \\ \neg\left(R^{\leq n}(t) \cap \left(T_{Trace}(\mathcal{F}) \setminus M'\right) \neq \emptyset\right) \\ \Leftrightarrow \\ \neg\left(\bigvee_{0 \leq i \leq n} \left(R^i(t) \cap \left(T_{Trace}(\mathcal{F}) \setminus M'\right) \neq \emptyset\right)\right) \\ \Leftrightarrow \\ \text{par (5.1)} \\ \neg\left(\bigvee_{0 \leq i \leq n} t \in \left(R^{-1}\right)^i \left(T_{Trace}(\mathcal{F}) \setminus M'\right)\right) \\ \Leftrightarrow \\ \neg\left(t \in \left(R^{-1}\right)^{\leq n} \left(T_{Trace}(\mathcal{F}) \setminus M'\right)\right) \end{gathered}$$

On a donc :

$$M'' = T_{Trace}(\mathcal{F}) \setminus \left(R^{-1}\right)^{\leq n} \left(T_{Trace}(\mathcal{F}) \setminus M'\right).$$

L'ensemble $M' = \pi_\Gamma^{-1}(M)$ est régulier car la projection inverse préserve la régularité et l'ensemble $T_{Trace}(\mathcal{F}) \setminus M'$ est régulier car la complémentation préserve la régularité.

Les ensembles $\left(R^{-1}\right)^i \left(T_{Trace}(\mathcal{F}) \setminus \pi_\Gamma^{-1}(M)\right)$ sont réguliers par rationalité de R^{-1}, de même que leur union pour $1 \leq i \leq n$, et le complément de leur union. Donc M'' est bien régulier. □

Théorème 58. *Soit R une relation d'abstraction, telle que R et R^{-1} soient rationnelles. Il existe une procédure de détection décidant si L exhibe M, pour tout ensemble régulier de traces L et pour tout comportement abstrait régulier M qui a la propriété de (m, n)-complétude pour des entiers m et n.*

Démonstration. En notant M'' l'ensemble des traces n-exhibant M, la propriété de (m, n)-complétude peut être reformulée de la façon suivante :

$$\begin{gathered} L \Cap M \\ \Leftrightarrow \\ \exists t' \in R^{\leq m}(L),\, t' \in M''. \\ \Leftrightarrow \\ R^{\leq m}(L) \cap M'' \neq \emptyset. \end{gathered}$$

M'' est régulier par le Lemme 57. La relation R est rationnelle donc elle préserve la régularité. Or L est régulier, donc $R^{\leq m}(L)$ est régulier également. Enfin, le test du vide de l'intersection de deux ensembles réguliers est décidable. □

En utilisant alors l'ensemble des traces n-exhibant M, on obtient la complexité suivante du problème de la détection. Cette complexité est linéaire en la taille de l'automate de traces du programme, une amélioration majeure sur la borne de complexité exponentielle du problème de la détection dans [66].

Théorème 59. *Soit R une relation d'abstraction définie par un système d'abstraction dont l'ensemble des instances de membres droits de ses règles est reconnu par un automate d'arbres A_R. Soit M un comportement abstrait régulier ayant la propriété de (m, n)-complétude et A_M un automate d'arbres reconnaissant l'ensemble des traces n-exhibant M par rapport à R.*

Il existe une procédure de détection décidant si un ensemble régulier de traces L, reconnu par un automate d'arbres A, exhibe M, en temps et espace $O\left(|A_R|^{m\cdot(m+1)/2} \times |A| \times |A_M|\right)$.

Démonstration. Notons M'' l'ensemble des traces n-exhibant M par rapport à R. La preuve du Théorème 58 reposait sur le résultat suivant :

$$\begin{gathered} L \Cap M \\ \Leftrightarrow \\ R^{\leq m}(L) \cap M'' = \emptyset \end{gathered}$$

Par le Théorème 71 sur la complexité de l'abstraction, que nous donnerons en Section 5.5, il existe un automate d'arbres reconnaissant $R^{\leq m}(L)$ de taille $O\left(|A_R|^{m\cdot(m+1)/2} \times |A|\right)$. L'intersection de deux automates d'arbres A_1 et A_2 produit un automate de taille $O(|A_1| \times |A_2|)$. Finalement, décider si un automate reconnaît l'ensemble vide prend un temps et un espace linéaire en sa taille. □

5.5 Abstraction de Traces

Relation d'Abstraction

Comme dit précédemment, l'abstraction d'une trace par rapport à un behavior pattern donné revient à transformer la trace lorsqu'elle contient une occurrence du behavior pattern, en insérant un symbole de Γ dans la trace. Ce symbole, appelé symbole d'abstraction, est inséré à la position après laquelle la fonctionnalité du behavior pattern est réalisée. Cette position est la plus logique pour coller à la sémantique de la trace, d'autant que lorsque deux behavior patterns apparaissent entrelacés, cette position permet de représenter l'ordre dans lequel leurs fonctionnalités sont réalisées, comme le montre l'exemple suivant.

Exemple 60. Considérons un behavior pattern décrivant la lecture d'un fichier sensible $\lambda_{read} := ReadFile$ et un behavior pattern décrivant l'envoi de données sur le réseau $\lambda_{send} := socket \cdot sendto$. La trace $socket \cdot ReadFile \cdot sendto$ sera jugée suspecte uniquement si le symbole d'abstraction λ_{read} est inséré immédiatement après $ReadFile$ et si le symbole d'abstraction λ_{send} est inséré après $sendto$, produisant la trace abstraite $socket \cdot ReadFile \cdot \lambda_{read} \cdot sendto \cdot \lambda_{send}$. En effet, dans ce cas la trace sera interprétée comme la lecture d'un fichier sensible suivie d'une communication réseau. Si le symbole d'abstraction λ_{read} pour $ReadFile$ avait été inséré plus loin dans la trace, par exemple après λ_{send}, nous aurions perdu la séquence "lecture-envoi". Le choix de cette position d'insertion est donc importante pour réduire les taux d'erreurs (faux positifs comme faux négatifs) de l'algorithme de détection.

Rappelons que, comme nous l'avons indiqué dans nos objectifs au début de ce chapitre, nous souhaitons éviter de remplacer les occurrences des behavior patterns par leurs symboles d'abstraction, afin de correctement abstraire des occurrences de behavior patterns entrelacés. Pour cette raison, les occurrences sont préservées lors de l'abstraction.

Considérons maintenant l'exemple suivant.

Exemple 61. Soit $x, x' \in X$ deux variables de sorte *Data* et y une variable de sorte *Trace*. L'abstraction du ping dans l'Exemple 44 est réalisée par réécriture en utilisant la règle

$$A_1(x, x') \cdot B_1(x, x') \cdot y \to A_1(x, x') \cdot \lambda(x) \cdot B_1(x, x') \cdot y$$

avec $A_1(x, x') = socket(x, \alpha) \cdot (T_{Trace}(\mathcal{F}_\Sigma) \setminus (T_{Trace}(\mathcal{F}_\Sigma) \cdot closesocket(x) \cdot T_{Trace}(\mathcal{F}_\Sigma))) \cdot sendto(x, \beta, x')$ et $B_1(x, x') = \{\epsilon\}$, et la règle

$$A_2(x) \cdot B_2(x) \cdot y \to A_2(x) \cdot \lambda(x) \cdot B_2(x) \cdot y$$

avec $A_2(x) = \{IcmpSendEcho(x)\}$ et $B_2(x) = \{\epsilon\}$.

On définit donc la relation d'abstraction en décomposant le behavior pattern en une union finie de concaténations d'ensembles $A_i(X)$ et $B_i(X)$ tels que les traces de $A_i(X)$ se terminent par l'action réalisant effectivement la fonctionnalité du behavior pattern. Ces ensembles $A_i(X)$ et $B_i(X)$ sont composés de traces concrètes uniquement, puisque les actions abstraites qui pourraient apparaître dans une trace partiellement réécrite ne sont pas censées influencer l'abstraction.

Définition 62. Soit $\lambda \in \Gamma$ un symbole d'abstraction, X un ensemble de variables de sorte *Data*, $\overline{x}$ une séquence de variables de X et y une variable de sorte *Trace*. Un *système d'abstraction* sur $T_{Trace}(\mathcal{F}, X \cup \{y\})$ est un ensemble fini de règles de réécriture de la forme :

$$A_i(X) \cdot B_i(X) \cdot y \to A_i(X) \cdot \lambda(\overline{x}) \cdot B_i(X) \cdot y$$

où les ensembles $A_i(X)$ et $B_i(X)$ sont des ensembles de traces concrètes de $T_{Trace}(\mathcal{F}_\Sigma, X)$.

Remarque 63. L'Exemple 61 définit une unique règle d'abstraction pour chaque manière de réaliser la fonctionnalité du behavior pattern mais il peut arriver qu'une réalisation soit associée à plus d'une règle d'abstraction, en particulier lorsque le behavior pattern exprime des contraintes complexes. L'exemple suivant illustre un tel cas.

Soit B un behavior pattern construit à partir d'une trace $a(x) \cdot b \cdot c(x)$ tel que la trace $d(x) \cdot e(x)$ libère la ressource x et soit donc interdite entre $a(x)$ et $c(x)$.

Définissons la notation suivante :

$$T_{a_1 \cdots a_n}(\mathcal{F}) = T_{Trace}(\mathcal{F}) \cdot a_1 \cdot T_{Trace}(\mathcal{F}) \cdots a_n \cdot T_{Trace}(\mathcal{F}).$$

On définit alors B par l'ensemble suivant :

$$B = \bigcup_{\sigma} \left((a(x) \cdot T_b(\mathcal{F}) \cdot c(x))\, \sigma \setminus T_{(d(x) \cdot e(x))\sigma}(\mathcal{F}) \right).$$

Supposons que l'action b réalise effectivement la fonctionnalité du behavior pattern : l'abstraction par rapport à B consiste alors à insérer une action abstraite $\lambda(x)$ immédiatement après l'action b.

Les traces réalisant le behavior pattern sont des traces vérifiant l'une des conditions suivantes :

- $d(x)$ apparaît entre $a(x)$ et b, et $e(x)$ n'apparaît pas entre $d(x)$ et $c(x)$;
- $e(x)$ apparaît entre b et $c(x)$, et $d(x)$ n'apparaît pas entre $a(x)$ et $e(x)$;
- $d(x)$ n'apparaît pas entre $a(x)$ et b, et $e(x)$ n'apparaît pas entre b et $c(x)$.

Ainsi, on définit trois règles de réécriture en utilisant les ensembles $A_i(x)$ et $B_i(x)$ suivants, correspondant respectivement aux trois cas ci-dessus :

- Pour la première condition, si $d(x)$ apparaît entre $a(x)$ et b, alors $e(x)$ n'apparaît pas entre $d(x)$ et $c(x)$ ssi $d(x) \cdot e(x)$ n'apparaît pas entre $a(x)$ et b, et $e(x)$ n'apparaît pas entre b et $c(x)$, ce qui est exprimé par :
  - $A_1(x) = \left(a(x) \cdot T_{d(x)}(\mathcal{F}) \cdot b\right) \setminus T_{d(x) \cdot e(x)}(\mathcal{F})$;
  - $B_1(x) = (T(\mathcal{F}) \cdot c(x)) \setminus T_{e(x)}(\mathcal{F})$.
- Pour la deuxième condition, si $e(x)$ apparaît entre b et $c(x)$, alors $d(x)$ n'apparaît pas entre $a(x)$ et $e(x)$ ssi $d(x) \cdot e(x)$ n'apparaît pas entre b et $c(x)$, et $d(x)$ n'apparaît pas entre $a(x)$ et b, ce qui est exprimé par :
  - $A_2(x) = (a(x) \cdot T(\mathcal{F}) \cdot b) \setminus T_{d(x)}(\mathcal{F})$;
  - $B_2(x) = \left(T_{e(x)}(\mathcal{F}) \cdot c(x)\right) \setminus T_{d(x) \cdot e(x)}(\mathcal{F})$.
- La dernière condition est exprimée par :
  - $A_3(x) = (a(x) \cdot T(\mathcal{F}) \cdot b) \setminus T_{d(x)}(\mathcal{F})$;
  - $B_3(x) = (T(\mathcal{F}) \cdot c(x)) \setminus T_{e(x)}(\mathcal{F})$.

Le système d'abstraction contient alors les trois règles suivantes :

$$\begin{aligned} A_1(x) \cdot B_1(x) \cdot y &\to A_1(x) \cdot \lambda(x) \cdot B_1(x) \cdot y \\ A_2(x) \cdot B_2(x) \cdot y &\to A_2(x) \cdot \lambda(x) \cdot B_2(x) \cdot y \\ A_3(x) \cdot B_3(x) \cdot y &\to A_3(x) \cdot \lambda(x) \cdot B_3(x) \cdot y. \end{aligned}$$

Le système de règles de réécriture que nous utilisons génère une relation de réduction sur $T_{Trace}(\mathcal{F})$ telle que le filtrage ne tienne pas compte des abstractions déjà faites dans la trace, c'est-à-dire travaille sur des traces projetées sur Σ.

Définition 64. La *relation de réduction* sur $T_{Trace}(\mathcal{F})$ générée par un système de n règles de réécriture $A_i(X) \cdot B_i(X) \cdot y \to A_i(X) \cdot \lambda(\overline{x}) \cdot B_i(X) \cdot y$ est la relation de réécriture $\to_{\mathcal{R}}$ telle que, pour tout $t, t' \in T_{Trace}(\mathcal{F}), t \to_{\mathcal{R}} t'$ ssi :

$$\begin{gathered}\exists \sigma \in Inst_X, \exists p \in Pos(t), \exists i \in [1..n], \\ \exists a \in T_{Trace}(\mathcal{F}) \cdot T_{Action}(\mathcal{F}_\Sigma), \exists b \in T_{Trace}(\mathcal{F}), \\ a|_\Sigma \in A_i(X)\sigma, b|_\Sigma \in B_i(X)\sigma, \\ t|_p = a \cdot b \cdot y\sigma \text{ et } t' = t[a \cdot \lambda(\overline{x})\sigma \cdot b \cdot y\sigma]_p.\end{gathered}$$

Une relation d'abstraction par rapport à un behavior pattern donné est alors la relation de réduction d'un système d'abstraction, dont les membres gauches de règles couvrent l'ensemble des traces réalisant la fonctionnalité du behavior pattern.

Définition 65. Soit B un behavior pattern associé à un symbole d'abstraction $\lambda \in \Gamma$. Soit X un ensemble de variables de sorte *Data* et y une variable de sorte *Trace*. Une *relation d'abstraction* par rapport à ce behavior pattern est la relation de réduction sur $T_{Trace}(\mathcal{F})$ générée par un système d'abstraction composé de n règles $A_i(X) \cdot B_i(X) \cdot y \to A_i(X) \cdot \lambda(\overline{x}) \cdot B_i(X) \cdot y$ vérifiant :

$$B = \bigcup_{i \in [1..n]} \bigcup_{\sigma \in Inst_X} (A_i(X) \cdot B_i(X))\sigma.$$

Finalement, on généralise la définition de l'abstraction à un ensemble de behavior patterns.

Définition 66. Soit C un ensemble fini de behavior patterns. Une *relation d'abstraction* par rapport à C est l'union des relations d'abstractions associées aux behavior patterns de C.

Abstraction Saine

La relation d'abstraction définie en Définition 65 autorise l'abstraction répétée de la même occurrence. Par exemple, si R est une relation d'abstraction par rapport au behavior $\lambda := a$, la trace a s'abstrait par R en $a \cdot \lambda$ mais aussi, en deux coups d'abstraction, en $a \cdot \lambda \cdot \lambda$, etc. Non seulement, ce type d'abstraction ne correspond pas à l'intuition mais il n'est pas souhaité car, même si en

pratique la (m, n)-complétude nous affranchit du calcul des formes normales, la définition même de l'exhibition d'un comportement repose sur l'existence de formes normales pour toute trace. Or l'existence de formes normales n'est pas assurée si une occurrence peut être réabstraite un nombre quelconque de fois.

À partir d'une relation d'abstraction donnée, on définit donc une notion de relation d'abstraction saine, qui requiert que la même action abstraite ne puisse pas être insérée deux fois après la même action concrète. En d'autres termes, si un terme $t = t_1 \cdot t_2$ est abstrait en un terme $t' = t_1 \cdot \alpha \cdot t_2$, où α est l'action abstraite insérée, alors si t_2 débute par une séquence d'actions abstraites, α n'apparaît pas dans cette séquence.

Définition 67. La *relation d'abstraction saine* pour une relation d'abstraction R est la relation R' définie par :

$$\begin{gathered}\forall t_1, t_2 \in T_{Trace}(\mathcal{F}), \forall \alpha \in T_{Action}(\mathcal{F}_\Gamma), \\ t_1 \cdot t_2 \rightarrow_{R'} t_1 \cdot \alpha \cdot t_2 \\ \Leftrightarrow \\ t_1 \cdot t_2 \rightarrow_R t_1 \cdot \alpha \cdot t_2 \\ \text{et} \\ \nexists u \in T_{Trace}(\mathcal{F}_\Gamma), \nexists u' \in T_{Trace}(\mathcal{F}), t_2 = u \cdot \alpha \cdot u'.\end{gathered}$$

En utilisant la définition ci-dessus, une occurrence de behavior pattern ne peut être abstraite qu'une seule fois.

Observons de plus que l'abstraction ne crée pas de nouvelles possibilités d'abstraction et que la relation R' est alors terminante, garantissant ainsi l'existence de formes normales pour toute trace.

Remarque 68. Notons qu'une relation d'abstraction saine par rapport à un ensemble de behavior patterns n'est pas confluente en général. Nous pourrions adapter la définition de la relation d'abstraction pour la rendre confluente, en définissant par exemple un ordre sur l'ensemble $T_{Action}(\mathcal{F}_\Gamma)$. Cependant, comme nous l'avons vu en Section 5.3, la détection opère sur l'ensemble des formes normales. Donc l'existence de plusieurs formes normales pour une trace ne compromet pas son mécanisme.

Abstraction Rationnelle

En pratique, un behavior pattern est régulier, de même que l'ensemble des instances des membres droits de ses règles d'abstraction. On montre que c'est suffisant pour garantir que la relation d'abstraction soit réalisable par un transducteur d'arbres, autrement dit qu'elle soit une transduction rationnelle

d'arbres. Le formalisme des transducteurs d'arbres est choisi pour ses propriétés formelles (clôture par union, composition, préservation de la régularité) et calculatoires intéressantes.

La preuve de la rationalité de l'abstraction repose sur la définition et le lemme suivants.

Définition 69. Soit Ω un ensemble de symboles de fonction de profil $Data^n \to Action$, $n \in \mathbb{N}$. Soit $L \subseteq T_{Trace}(\mathcal{F})$ un ensemble de traces. La forme Ω-généralisée de L, notée $\amalg_\Omega(L)$, est l'ensemble :

$$\begin{aligned}\amalg_\Omega(L) = \{ \quad & t_0 \cdot a_1 \cdot t_1 \cdots a_n \cdot t_n \mid \\ & a_1, \ldots, a_n \in T_{Action}(\Omega \cup \mathcal{F}_d), \\ & t_0 \cdots t_n \in L\}.\end{aligned}$$

Lemme 70. *Soit $\Omega \subseteq \mathcal{F}_a$ un ensemble de symboles de fonction de profil $Data^n \to Action$, $n \in \mathbb{N}$. Si A est un automate d'arbres, alors il existe un automate d'arbres de taille $O(|A|)$ reconnaissant la forme Ω-généralisée de $\mathcal{L}(A)$.*

Démonstration. On définit un transducteur d'arbres $\tau = (\mathcal{F}, \mathcal{F} \cup \Omega, \{q_t, q_a, q_d\}, q_t, \Delta)$ tel que : $R_\tau(\mathcal{L}(A)) = \amalg_\Omega(\mathcal{L}(A))$.

Δ est composé des règles suivantes :

- $q_t(\cdot(x_1, x_2)) \to \cdot(q_a(x_1), q_t(x_2))$;
- $q_t(\epsilon) \to \epsilon$;
- Pour tout $k \in \mathbb{N}$, pour tout $f \in \mathcal{F}_a^{(k)}$, Δ contient une règle :
  $q_a(f(x_1, \ldots, x_k)) \to f(q_d(x_1), \ldots, q_d(x_k))$;
- Pour tout $d \in \mathcal{F}_d$, Δ contient une règle :
  $q_d(d) \to d$;
- Pour tout terme $\alpha \in T_{Action}(\mathcal{F}_\Omega)$, Δ contient une règle :
  $q_t(x) \to \cdot(\alpha, q_t(x))$.

Le transducteur τ réalise $\amalg_\Omega$ et a une taille constante par rapport à la taille de A. Le Lemme 39 entraîne donc que $\amalg_\Omega(\mathcal{L}(A))$ est reconnu par un automate d'arbres de taille $O(|A| \times |\tau|) = O(|A|)$. □

Théorème 71. *Soit B un behavior pattern et R une relation d'abstraction saine par rapport à B, définie par un système d'abstraction dont l'ensemble des instances de membres droits de ses règles est reconnu par un automate d'arbres A_R.*

Alors R et R^{-1} sont rationnelles et, pour tout automate d'arbres A, $R(\mathcal{L}(A))$ est reconnu par un automate d'arbres de taille $O(|A| \cdot |A_R|)$.

Démonstration. On construit deux transducteurs d'arbres réalisant R et R^{-1}.

Soit n le nombre de règles du système d'abstraction. Soit C l'ensemble des instances de membres droits de règles de R :

$$C = \bigcup_{i \in [1..n]} \bigcup_{\sigma \in Inst_{X \cup \{y\}}} A_i(X)\,\sigma \cdot \lambda(\overline{x})\,\sigma \cdot B_i(X)\,\sigma \cdot y\sigma \subseteq T_{Trace}(\mathcal{F}).$$

Soit $\diamondsuit$ une constante de sorte *Action* qui n'est pas dans $\mathcal{F}_a$. On considère l'ensemble suivant :

$$Img_{\diamondsuit}(R) = \{t_1 \cdot \diamondsuit \cdot \alpha \cdot \diamondsuit \cdot t_2 \mid \begin{array}{l} t_1, t_2 \in T_{Trace}(\mathcal{F}), \\ \alpha \in T_{Action}(\mathcal{F}_\Gamma), \\ (t_1 \cdot t_2, t_1 \cdot \alpha \cdot t_2) \in R\}. \end{array}$$

On va montrer que cet ensemble est reconnu par un automate d'arbres, de taille $O(|A_R|)$, et on va alors utiliser cet ensemble pour construire un transducteur d'arbres reconnaissant R et R^{-1}.

Définissons l'ensemble C' par :

$$\begin{array}{c} C' = \amalg_{\{\diamondsuit\}}(C) \\ \cap \\ (T_{Trace}(\mathcal{F}) \cdot \diamondsuit \cdot T_{Action}(\mathcal{F}_\Gamma) \cdot \diamondsuit \cdot T_{Trace}(\mathcal{F})). \end{array}$$

Par le Lemme 70 appliqué à $\Omega = \{\diamondsuit\}$ et à l'automate A_R, l'ensemble $\amalg_{\{\diamondsuit\}}(C)$ est régulier et reconnu par un automate d'arbres, de taille $O(|A_R|)$. L'ensemble $T_{Trace}(\mathcal{F}) \cdot \diamondsuit \cdot T_{Action}(\mathcal{F}_\Gamma) \cdot \diamondsuit \cdot T_{Trace}(\mathcal{F})$ est également reconnu par un automate d'arbres de taille constante. Donc leur intersection C' est régulière et reconnue par un automate d'arbres $A_{C'}$ de taille $O(|A_R|)$. Les termes de C' sont les termes de C où l'action abstraite a été isolée entre deux diamants.

Définissons maintenant l'ensemble C'' par :

$$\begin{array}{c} C'' = \amalg_{\Gamma}(C') \\ \cap \\ (T_{Trace}(\mathcal{F}) \cdot T_{Action}(\mathcal{F}_\Sigma) \cdot \\ \diamondsuit \cdot T_{Action}(\mathcal{F}_\Gamma) \cdot \diamondsuit \cdot T_{Trace}(\mathcal{F})). \end{array}$$

Par le Lemme 70 appliqué à $\Omega = \Gamma$ et à l'automate $A_{C'}$, l'ensemble $\amalg_{\Gamma}(C')$ est régulier et reconnu par un automate d'arbres de taille $O(|A_R|)$. L'ensemble $T_{Trace}(\mathcal{F}) \cdot T_{Action}(\mathcal{F}_\Sigma) \cdot \diamondsuit \cdot T_{Action}(\mathcal{F}_\Gamma) \cdot \diamondsuit \cdot T_{Trace}(\mathcal{F})$ est également reconnu par un automate d'arbres qui est de taille constante. Donc leur intersection C'' est régulière et reconnue par un automate d'arbres $A_{C''}$ de taille $O(|A_R|)$.

Finalement, on retire de C'' les termes qui violent la condition de terminaison de la Définition 67. On définit l'ensemble C''' par :

$$C''' = C'' \setminus \bigcup_{\alpha \in T_{Action}(\mathcal{F}_\Gamma)} (T_{Trace}(\mathcal{F}) \cdot \diamondsuit \cdot \alpha \cdot \diamondsuit \cdot T_{Trace}(\mathcal{F}_\Gamma) \cdot \alpha \cdot T_{Trace}(\mathcal{F})).$$

L'ensemble $T_{Action}(\mathcal{F}_\Gamma)$ est fini, donc l'ensemble $\bigcup_{\alpha \in T_{Action}(\mathcal{F}_\Gamma)} T_{Trace}(\mathcal{F}) \cdot \diamondsuit \cdot \alpha \cdot \diamondsuit \cdot T_{Trace}(\mathcal{F}_\Gamma) \cdot \alpha \cdot T_{Trace}(\mathcal{F})$ est reconnu par un automate d'arbres de taille constante. L'ensemble C''' est donc régulier et reconnu par un automate d'arbres de taille $O(|A_{C''}|) = O(|A_R|)$, et il vérifie :

$$T_{Trace}(\mathcal{F}) \cdot C''' = Img_\diamondsuit(R). \tag{5.2}$$

En effet, pour tout $t_1, t_2 \in T_{Trace}(\mathcal{F})$ et pour tout $\alpha \in T_{Action}(\mathcal{F}_\Gamma)$:

$$\begin{gathered}
(t_1 \cdot t_2, t_1 \cdot \alpha \cdot t_2) \in R \\
\Leftrightarrow \\
\exists u_1, v_1 \in T_{Trace}(\mathcal{F}), \\
t_1 = u_1 \cdot v_1 \\
v_1|_\Sigma \cdot \alpha \cdot t_2|_\Sigma \in C, \\
v_1 \in T_{Trace}(\mathcal{F}) \cdot T_{Action}(\mathcal{F}_\Sigma), \\
t_2 \notin T_{Trace}(\mathcal{F}_\Gamma) \cdot \alpha \cdot T_{Trace}(\mathcal{F}) \\
\Leftrightarrow \\
\exists u_1, v_1 \in T_{Trace}(\mathcal{F}), \\
t_1 = u_1 \cdot v_1, \\
v_1|_\Sigma \cdot \diamondsuit \cdot \alpha \cdot \diamondsuit \cdot t_2|_\Sigma \in C', \\
v_1 \in T_{Trace}(\mathcal{F}) \cdot T_{Action}(\mathcal{F}_\Sigma), \\
t_2 \notin T_{Trace}(\mathcal{F}_\Gamma) \cdot \alpha \cdot T_{Trace}(\mathcal{F}) \\
\Leftrightarrow \\
\exists u_1, v_1 \in T_{Trace}(\mathcal{F}), \\
t_1 = u_1 \cdot v_1, \\
v_1 \cdot \diamondsuit \cdot \alpha \cdot \diamondsuit \cdot t_2 \in C'', \\
t_2 \notin T_{Trace}(\mathcal{F}_\Gamma) \cdot \alpha \cdot T_{Trace}(\mathcal{F}) \\
\Leftrightarrow \\
\exists u_1, v_1 \in T_{Trace}(\mathcal{F}), \\
t_1 = u_1 \cdot v_1, \\
v_1 \cdot \diamondsuit \cdot \alpha \cdot \diamondsuit \cdot t_2 \in C''' \\
\Leftrightarrow \\
t_1 \cdot \diamondsuit \cdot \alpha \cdot \diamondsuit \cdot t_2 \in T_{Trace}(\mathcal{F}) \cdot C'''.
\end{gathered}$$

On définit maintenant les transducteurs réalisant R et R^{-1}. Pour cela, considérons la relation T définie par :

$$\begin{aligned}
T = \{ \quad & (t \cdot t', t \cdot \diamondsuit \cdot \alpha \cdot \diamondsuit \cdot t'), \mid \\
& t, t' \in T_{Trace}(\mathcal{F}), \alpha \in T_{Action}(\mathcal{F}_\Gamma) \ \}.
\end{aligned}$$

Clairement, les relations T et T^{-1} sont rationnelles et reconnues par des transducteurs τ_T et $\tau_{T^{-1}}$ de taille constante.

L'ensemble $T_{Trace}(\mathcal{F}) \cdot C'''$ est reconnu par un automate d'arbres de taille $O(|A_R|)$, donc il existe un transducteur d'arbres $\tau_{C'''}$ réalisant la relation $\{(t,t) \mid t \in T_{Trace}(\mathcal{F}) \cdot C'''\}$ et de taille $O(|A_R|)$.

De plus, soit $\tau_\diamondsuit$ le transducteur d'arbres sur $T_{Trace}(\mathcal{F} \cup \{\diamondsuit\})$ qui réalise la projection sur $\Sigma \cup \Gamma$, i.e. qui supprime les diamants, et $\tau_{\diamondsuit^{-1}}$ le transducteur d'arbres sur $T_{Trace}(\mathcal{F} \cup \{\diamondsuit\})$ qui insère des diamants aléatoirement dans la sortie. $\tau_\diamondsuit$ et $\tau_{\diamondsuit^{-1}}$ sont de taille constante.

Alors, R est réalisé par le transducteur d'arbres $\tau_\diamondsuit \circ \tau_{C'''} \circ \tau_T$. En effet, pour tout $t_1, t_2 \in T_{Trace}(\mathcal{F}), \alpha \in T_{Action}(\mathcal{F}_\Gamma)$, soit $t' = t_1 \cdot \diamondsuit \cdot \alpha \cdot \diamondsuit \cdot t_2$, on a :

$$\begin{gathered}
(t_1 \cdot t_2, t_1 \cdot \alpha \cdot t_2) \in R \\
\Leftrightarrow \\
t' = t_1 \cdot \diamondsuit \cdot \alpha \cdot \diamondsuit \cdot t_2 \in Img_\diamondsuit(R) \\
\Leftrightarrow \\
\text{par } (5.2) \\
t' \in T_{Trace}(\mathcal{F}) \cdot C''' \\
\Leftrightarrow \\
(t_1 \cdot t_2, t') \in T, (t', t') \in R_{\tau_{C'''}} \text{ and } (t', t_1 \cdot \alpha \cdot t_2) \in R_{\tau_\diamondsuit} \\
\Leftrightarrow \\
(t_1 \cdot t_2, t_1 \cdot \alpha \cdot t_2) \in R_{\tau_\diamondsuit \circ \tau_{C'''} \circ \tau_T}.
\end{gathered}$$

De façon similaire, R^{-1} est réalisé par le transducteur d'arbres $\tau_{T^{-1}} \circ \tau_{C'''} \circ \tau_{\diamondsuit^{-1}}$. En effet : $\left(R_{\tau_\diamondsuit}\right)^{-1} = R_{\tau_{\diamondsuit^{-1}}}$ et $\left(R_{\tau_{C'''}}\right)^{-1} = R_{\tau_{C'''}}$, donc :

$$R^{-1} = R_{\tau_{T^{-1}} \circ \tau_{C'''} \circ \tau_{\diamondsuit^{-1}}}.$$

Enfin, pour tout automate de traces A, l'ensemble $R(\mathcal{L}(A))$ est égal à $\tau_\diamondsuit(\tau_{C'''}(\tau_T(\mathcal{L}(A))))$ et est reconnu par un automate de traces de taille $O(|A| \cdot |A_R|)$, par le Lemme 39. □

5.6 Application à la Détection d'une Fuite d'Information

L'abstraction peut être appliquée à la détection de menaces génériques, et en particulier à la détection d'une fuite d'informations sensibles. Une telle fuite peut être décomposée en deux étapes : capture des informations sensibles et envoi de ces informations vers un emplacement exogène. Les données capturées peuvent être les touches entrées au clavier, des mots de passe ou des données lues sur un emplacement réseau sensible, tandis que l'emplacement exogène peut être le réseau, un média amovible, etc. Ainsi, on définit un behavior pattern $\lambda_{steal}(x)$, qui représente la capture d'une donnée sensible x,

et un behavior pattern $\lambda_{leak}(x)$, qui représente la transmission de x vers un emplacement exogène quelconque. De plus, les données capturées devant rester valides jusqu'à leur envoi, on définit un behavior pattern $\lambda_{inval}(x)$, qui représente l'invalidation de ces données. Le comportement abstrait de fuite d'information est alors défini par :

$$M := \exists x.\, \lambda_{steal}(x) \wedge \neg\lambda_{inval}(x)\ \mathbf{U}\ \lambda_{leak}(x)\,.$$

En observant plusieurs échantillons de programmes malicieux, comme des keyloggers, des applications de fuite de SMS ou des applications mobiles dérobant des informations personnelles, on considère les définitions suivantes des trois behavior patterns impliqués :

- $\lambda_{steal}(x)$ décrit une fonctionnalité de capture d'une interaction clavier [1] et, sur des téléphones portables Android, d'obtention du numéro IMEI :

$$\begin{aligned}\lambda_{steal}(x) := {} & GetAsyncKeyState(x)\vee\\ & (RegisterDev(\texttt{KBD},\texttt{SINK})\odot GetInputData(x,\texttt{INPUT}))\vee\\ & (\exists y.\, SetWindowsHookEx(y,\texttt{WH_KEYBOARD_LL})\wedge\\ & \neg UnhookWindowsHookEx(y)\mathbf{U}\, HookCalled(y,x))\vee\\ & \exists y. TelephonyManager_getDeviceId(x,y).\end{aligned}$$

- $\lambda_{leak}(x)$ décrit une fonctionnalité d'envoi réseau sous Windows ou sous Android :

$$\begin{aligned}\lambda_{leak}(x) := {} & \exists y,z.\, sendto(z,x,y)\vee\\ & \exists y,z.\,(connect\,(z,y)\wedge\neg close(z)\,\mathbf{U}\, send(z,x))\vee\\ & \exists c,s.\, HttpURLConnection_getOutputStream(s,c)\wedge\\ & \neg OutputStream_close(s)\mathbf{U}\, OutputStream_write(s,x).\end{aligned}$$

- $\lambda_{inval}(x)$ décrit l'écrasement ou la libération de x :

$$\begin{aligned}\lambda_{inval}(x) := {} & free(x)\vee\exists y.\, sprintf_0(x,y)\vee\\ & GetInputData(x,\texttt{INPUT})\vee\ldots\end{aligned}$$

Pour terminer, les données capturées ne sont en général pas transmises dans leur forme brute. On prend donc en compte des transformations de ces données via le behavior pattern $\lambda_{depends}(x,y)$ qui décrit une dépendance de x sur y. Par exemple, x peut consister en une représentation de y par une chaîne de caractères, ou x peut consister en un chiffrement ou en un encodage de y :

$$\begin{aligned}\lambda_{depends}(x,y) := {} & sprintf_0(x,y)\vee\exists s.\, sprintf_1(x,s,y)\vee\\ & \exists sb.\, StringBuilder_append(sb,y)\odot SB_toString(x,sb).\end{aligned}$$

1. On suppose que l'exécution d'une hook f avec un argument x est représentée dans une trace par une action $HookCalled\,(f,x)$.

Pour alors tenir compte d'une telle transformation des données capturées, on adapte la définition du comportement abstrait de fuite d'information :

$$M := \exists x, y.\, \lambda_{steal}(x) \wedge \quad \begin{array}{l} \neg\lambda_{inval}(x)\ \mathbf{U}\ \lambda_{depends}(y, x) \wedge \\ \neg\lambda_{inval}(y)\ \mathbf{U}\ \lambda_{leak}(y)\,. \end{array}$$

Bien entendu, on peut adapter cette formule pour autoriser plus d'une transformation de données.

5.7 Expérimentations

À partir de cette définition formelle du comportement de fuite d'information et en nous fondant sur le formalisme d'abstraction décrit dans ce chapitre, nous avons réalisé des expériences sur des applications malicieuses dérobant des données sensibles.

Nous avons adopté une approche originale consistant à représenter un automate de traces par un processus algébrique et à utiliser ensuite la boîte à outils CADP pour réaliser à la fois l'abstraction de l'automate de traces, par synchronisation de processus, puis la détection d'un comportement de haut niveau, par simple model checking. Nous avons testé ce cadre d'expérimentations sur des codes malicieux incluant des applications mobiles dans le but de détecter le comportement de fuite de données décrit dans la section précédente.

Les échantillons de programmes malicieux ont été collectés dans des bases de données publiques, comme Offensive Computing [2] et Contagio Malware Dump [3].

Mise en Œuvre

Pour réaliser l'abstraction de behavior patterns et la détection de comportements de haut niveau en présence de données, nous avons utilisé la boîte à outils CADP [46], qui permet de manipuler et de vérifier des processus communicants spécifiés dans le langage LOTOS. Le choix du formalisme d'une algèbre de processus pour représenter des traces a plusieurs origines. D'une part, une action $f(d_1, \ldots, d_n)$ peut être représentée par une écriture sur une porte donnée, avec $n+1$ paramètres, $f, d_1, \ldots, d_n$: l'exécution d'un processus exposant cette porte simule alors une trace et l'ensemble des exécutions d'un processus exposant cette porte simule un ensemble de traces. D'autre part, il existe une

2. http://www.offensivecomputing.net
3. http://contagiodump.blogspot.com

correspondance naturelle entre le processus du programme et un processus algébrique. En outre, la représentation du parallélisme et de la synchronisation de processus permet de simuler les threads et les mécanismes de synchronisation fournis par les librairies du système. Enfin, la relation de transduction sous-jacente à l'abstraction peut être simulée de façon naturelle par une synchronisation de processus avec une porte d'entrée et une porte de sortie, comme nous allons le voir.

Par ailleurs, nous avons choisi plus particulièrement CADP pour deux raisons. D'une part, il contient un outil de vérification, EVALUATOR4, qui permet de vérifier des formules de logique temporelle exprimées dans le langage MCL [82], un fragment du mu-calcul modal étendu avec des variables. Ainsi, les formules FOLTL peuvent être exprimées naturellement dans le langage MCL. Et d'autre part, il met en œuvre plusieurs techniques d'optimisation : un processus peut être représenté explicitement (l'ensemble des états est pré-calculé) ou implicitement (l'ensemble des états est calculé à la demande, permettant d'optimiser les ressources nécessaires, en espace comme en temps) ; les espaces d'états sont condensés à l'aide de techniques de réduction par ordre partiel et de minimisation compositionnelle. Ainsi, notre mise en œuvre profite pleinement de ces optimisations puisque l'abstraction construit une représentation implicite de l'ensemble de traces partiellement abstraites et la vérification est effectuée par EVALUATOR4 (qui peut opérer directement sur une représentation implicite).

On représente donc l'automate de traces d'un programme par un système de processus communicants exprimé en LOTOS, avec une porte particulière, notée `api`, sur laquelle les communications correspondent à des appels de librairie. Ces communications sont paramétrées, ce qui permet de représenter les arguments de l'appel. Une interaction sur cette porte est de la forme `api !f !arg1 ... !argn` et représente l'appel de librairie $f(arg1, \ldots, argn)$. Par exemple, l'instruction Java `sb = new StringBuilder(s)` est représentée par une interaction de la forme `api !StringBuilder_new !sb !s`, où `api` est la porte, `StringBuilder_new` est un entier représentant l'appel de librairie et est associé à un symbole de Σ, `sb` et `s` sont des entiers représentant la valeur de retour et le paramètre de l'appel et sont associés à des symboles de $\mathcal{F}_d$.

Exemple 72. L'ensemble de traces $\{a(1) \cdot b, c \cdot d(2,3)\}$ est représenté par le processus LOTOS suivant :

```
process P [api]: exit :=
  api !a !1;
  api !b;
  exit
```

```
  []
  api !c;
  api !d !2 !3;
  exit
```

La notation `[]` représente un choix non déterministe.

Nous avons vu que l'abstraction est rationnelle, c'est-à-dire réalisable par un transducteur de traces. Toute transduction de traces peut être simulée par une synchronisation de processus en LOTOS.

Par exemple, supposons que nous souhaitions transformer l'ensemble de traces de l'exemple précédent de façon à insérer après une action $a(x)$ une action $e(x)$. Cette transformation est réalisable par un transducteur qui lit une action en entrée et l'écrit en sortie en insérant éventuellement l'action $e(x)$ si l'action lue était une instance de $a(x)$. Ce transducteur peut être représenté par un processus LOTOS, noté `T`, à deux portes : une porte d'entrée, c'est-à-dire `api`, et une porte de sortie, que nous notons `api′`.

Le processus T est défini de la façon suivante, et décomposé en fonction du nombre d'arguments de l'appel de librairie simulé par la porte `api` :

```
process T [api, api′]: exit :=
  api ?call:Nat [call <> a];
  api′ !call;
  T [api, api′]
  []
  api ?call:Nat ?arg1:Nat [call <> a];
  api′ !call !arg1;
  T [api, api′]
  []
  api ?call:Nat ?arg1:Nat ?arg2:Nat [call <> a];
  api′ !call !arg1 !arg2;
  T [api, api′]
  []
  api !a ?arg1:Nat;
  api′ !a !arg1;
  api′ !e !arg1;
  T [api, api′]
  []
  exit
```

On simule alors l'application de la transformation précédente à un ensemble de traces par la synchronisation, sur la porte `api`, du processus `P` (représentant l'ensemble de traces) avec le processus `T` (représentant le transducteur) : le processus résultant `P′` expose la porte `api′` sur laquelle on observe les traces transformées et représente l'ensemble de traces transformé.

```
process P′ [api′]: exit :=
  hide api in
    P [api] |[api]| T [api, api′]
```

La notation `hide api` permet de cacher les interactions sur la porte `api` dans le processus résultant. La notation `|[api]|` permet de synchroniser les deux processus sur la porte `api`.

L'abstraction étant rationnelle, on la simule alors en représentant le transducteur qui la réalise par un processus LOTOS puis on simule l'abstraction d'un ensemble de traces par une synchronisation de processus : le processus représentant les traces du programme est synchronisé sur la porte `api` avec un processus représentant le transducteur réalisant l'abstraction. Le processus résultant expose une porte `api′`, où les appels de librairie peuvent apparaître parsemés d'actions abstraites comme : `api′ !`λ` !arg`. Plus généralement, chaque behavior pattern est associé à un processus LOTOS : P_{λ_1}, P_{λ_2}... On représente la relation d'abstraction R par le processus suivant :

```
process P_R [api, api′]: exit :=
  P_λ1 [api, api′]
  []
  P_λ2 [api, api′]
  []
  ...
```

Puis, si un ensemble de traces L est représenté par un processus P_L, l'ensemble $L' = R^{\leq 2}(L)$ est représenté par le processus suivant :

```
process P_L′ [api]: exit :=
  P_L [api]
  []
  hide api′ in
    P_L [api′] |[api′]| P_R [api′, api]
  []
  hide api′, api″ in
    P_L [api″] |[api″]| P_R [api″, api′] |[api′]| P_R [api′, api]
```

De même, l'ensemble de traces $L' = R_{\lambda_2}(R_{\lambda_1}(L))$ est représenté par le processus suivant :

```
process P_L′ [api]: exit :=
  hide api′, api″ in
    P_L [api″] |[api″]| P_λ1 [api″, api′] |[api′]| P_λ2 [api′, api]
```

Ainsi, la détection dans un ensemble de traces L d'un comportement défini par une formule FOLTL φ_M et ayant la propriété de $(m, 0)$-complétude est réalisée de la façon suivante :

– Le processus représentant $R^{\leq m}(L)$ est construit avec CADP.
– Ce processus est ensuite soumis à EVALUATOR4 pour valider la formule φ_M.

De même, la détection d'un comportement défini par la formule FOLTL $\varphi_M = \exists Y.\, \lambda_1(\overline{x_1}) \wedge \neg(\exists Z.\, \lambda_2(\overline{x_2}))\, \mathbf{U}\, \lambda_3(\overline{x_3})$ et vérifiant les conditions du Théorème 54 est équivalente à :

$$\exists t \in\ R_{\lambda_2}\!\downarrow (R^{\leq 2}(L))\big|_{\Gamma}\,,\, t \models \varphi_M$$

et est réalisée de la façon suivante, lorsque la relation $R_{\lambda_2}\!\downarrow$ soit rationnelle :
– En simulant la relation de transduction $R_{\lambda_2}\!\downarrow$ par synchronisation de processus, on construit un processus représentant $R_{\lambda_2}\!\downarrow (R^{\leq 2}(L))$. La projection sur Γ est également simulée par synchronisation de processus, produisant un processus $R_{\lambda_2}\!\downarrow (R^{\leq 2}(L))\big|_{\Gamma}$.
– Ce processus est ensuite soumis à l'outil EVALUATOR4 de CADP pour valider la formule φ_M.

Détection du Comportement de Fuite d'Information

Considérons le comportement de fuite d'information défini en Section 5.6 du Chapitre 5 :

$$M := \exists x, y.\, \lambda_{steal}\,(x) \wedge \neg\lambda_{inval}\,(x)\ \mathbf{U}\, \lambda_{leak}\,(x, y)\,.$$

Remarque 73. Constatons que la relation $R_{\lambda_{inval}}\!\downarrow$ est rationnelle car le behavior pattern λ_{inval} est défini par des ensembles A_i et B_i tels que $B_i = \{\epsilon\}$ et A_i est composé de traces ne contenant qu'une action, comme $sprintf\,(s, f)$ ou $OutputStream_close\,(s)$. Ainsi, un transducteur peut être défini de telle sorte que lorsqu'il rencontre une telle action, il écrive l'action abstraite correspondante sur la sortie et continue l'abstraction. Notons que lorsqu'une même action peut être abstraite de plusieurs façons, le transducteur doit alors réaliser toutes les abstractions possibles, dans tous les ordres possibles. Par exemple, supposons que le système d'abstraction contienne les règles $a\,(x, y) \rightarrow a\,(x, y) \cdot \lambda_{invalidate}\,(x)$ et $a\,(x, y) \rightarrow a\,(x, y) \cdot \lambda_{invalidate}\,(y)$, alors le transducteur contiendra une règle $a\,(x, y) \rightarrow a\,(x, y) \cdot \lambda_{invalidate}\,(x) \cdot \lambda_{invalidate}\,(y)$ et une règle $a\,(x, y) \rightarrow a\,(x, y) \cdot \lambda_{invalidate}\,(y) \cdot \lambda_{invalidate}\,(x)$.

On définit en LOTOS un processus `ABSTRACT_SIG` simulant le calcul de l'ensemble $R_{\lambda_{inval}}\!\downarrow (R^{\leq 2}(L))\big|_{\Gamma}$:

```
process ABSTRACT_SIG [api, out]: exit :=
   hide api1, api2 in
```

```
        ABSTRACT_SIG_Rm [api, api1]
        |[api1]|
        ABSTRACT_SIG_Rnorm [api1, api2]
        |[api2]|
        PROJECT_ABSTRACT [api2, out]
endproc

process ABSTRACT_SIG_Rm [api, out]: exit :=
    hide api1 in
        BP_STEAL [api, api1]
        |[api1]|
        BP_LEAK [api1, out]
endproc

process ABSTRACT_SIG_Rnorm [api, out]: exit :=
    BP_INVAL_Total [api, out]
endproc
```

Les processus `ABSTRACT_SIG_Rm` et `ABSTRACT_SIG_Rnorm` calculent $R^{\leq m}(L)$ et $R_{\lambda_2}\downarrow(L)$. En fait, dans le cas du comportement M à détecter, on simplifie le processus `ABSTRACT_SIG_Rm` pour ne calculer que l'ensemble $R_{\lambda_{leak}}(R_{\lambda_{steal}}(L))$. Enfin, les processus `BP_STEAL` et `BP_LEAK` réalisent l'abstraction $R_{\lambda_{steal}}$ et $R_{\lambda_{leak}}$ respectivement et le processus `BP_INVAL_Total` réalise l'abstraction totale $R_{\lambda_{inval}}\downarrow$, en simulant le transducteur décrit en Remarque 73. Par exemple, l'abstraction du behavior pattern λ_{steal} lorsqu'il est réalisé par l'appel de librairie `GetRawInputData` (capture d'une interaction clavier) est réalisée par le processus suivant :

```
process BP_STEAL [api, out]: exit :=
    CONSUME_MANY[api, out] (_)
    >>
    REQ_GetRawInputData(hRawInput, pData, pData_header_dwType,
               pData_header_hDevice, pData_data_keyboard_Flags,
               pData_data_keyboard_VKey);
    HL_Steal(pData_data_keyboard_VKey);
    END_MATCHING[api, out] (HL_STEAL, pData_data_keyboard_VKey)
endproc
```

Le processus `CONSUME_MANY` simule le transducteur lisant un nombre arbitraire d'appels de librairie en les réécrivant tels quels sur la sortie ; le processus `END_MATCHING` simule le transducteur lisant un nombre arbitraire d'appels de librairie en les réécrivant tels quels sur la sortie et en s'assurant que la condition de terminaison est respectée (i.e. cette occurrence n'avait pas encore été abstraite). `REQ_GetRawInputData` et `HL_Steal` sont deux macros simulant res-

pectivement la lecture d'une action concrète $GetRawInputData\,(\ldots)$ sur la porte `api` et l'écriture d'une action abstraite $\lambda_{steal}\,(data)$ sur la porte `out` :

```
#define REQ_GetRawInputData(hRawInput, pData, pData_header_dwType,\
          pData_header_hDevice, pData_data_keyboard_Flags,\
          pData_data_keyboard_VKey)\
       api !GETRAWINPUTDATA ?hRawInput:Nat ?pData:Nat\
          ?pData_header_dwType:Nat ?pData_header_hDevice:Nat\
          ?pData_data_keyboard_Flags:Nat ?pData_data_keyboard_VKey:Nat;\
       out !GETRAWINPUTDATA !hRawInput !pData !pData_header_dwType\
          !pData_header_hDevice !pData_data_keyboard_Flags\
          !pData_data_keyboard_VKey

#define HL_Steal(data)\
       out !HL_STEAL !data
```

Pour terminer, le comportement de fuite de données M est défini dans le langage MCL attendu par EVALUATOR4 par la formule suivante :

```
< true* . {api !λ_steal ?data:Nat} >
  < (not({api !λ_invalidate !data}))* .
   {api !λ_leak ?dest:Nat !data} > true
```

Ainsi, pour décider si $L \Cap M$, on a simulé la construction de l'ensemble $R_{\lambda_{inval}}\!\downarrow (R^{\leq 2}(L))\big|_{\Gamma}$ en synchronisant le processus représentant l'ensemble de traces L avec le processus `ABSTRACT_SIG`, puis on a validé la formule MCL représentant φ_M sur le processus résultant.

Dans un second temps, considérons le cas plus général du comportement de fuite d'information tenant compte de la transformation des données avant leur fuite. Dans le cas d'une seule transformation, on avait :

$$M := \exists x, y.\, \lambda_{steal}\,(x) \wedge \neg\lambda_{inval}\,(x)\ \mathbf{U}\ \lambda_{depends}\,(y, x) \wedge \neg\lambda_{inval}\,(y)\ \mathbf{U}\ \lambda_{leak}\,(y)\,.$$

La détection est alors réalisée de la même manière que pour le comportement sans transformations, à une exception près : les behavior patterns $\lambda_{depends}$ et λ_{inval} doivent être indépendants, un pré-requis du Théorème 54. Or une action $sprintf\,(x, y)$ peut être interprétée à la fois comme une dépendance de x sur y et comme une invalidation de x : les behavior patterns $\lambda_{depends}$ et λ_{inval} ne sont donc pas indépendants. Pour résoudre ce problème, l'automate de traces est transformé, préalablement à la détection : une action $depends_invalidate(x)$, indiquant une invalidation du paramètre x est insérée avant toute action pouvant être interprétée à la fois comme une invalidation et une dépendance. Par exemple, on insère une action $depends_invalidate\,(x)$ avant toute action $sprintf\,(x, y)$. On modifie alors le behavior pattern λ_{inval}

pour ne plus prendre en compte l'action $sprintf(x, y)$ mais seulement l'action $depends_invalidate\,(x)$. Ainsi, l'indépendance des behavior patterns est bien garantie.

Nous avons testé la validité de notre approche sur plusieurs types de malwares réalisant une fuite d'informations : des applications de keylogging capturant les caractères entrés au clavier par l'appel `RegisterRawInputDevices` ou par l'installation d'une hook (enregistrement d'une fonction à appeler à chaque interaction du clavier), une application Android transmettant vers un numéro tiers tous les SMS reçus et envoyés, une application Android récupérant des informations personnelles d'un téléphone portable (numéro IMEI, ...).

En généralisant à un nombre borné quelconque de transformations des données, nous avons ainsi pu détecter avec succès le comportement de fuite d'information dans les échantillons testés.

5.8 Conclusion

Dans ce chapitre, nous avons présenté une approche originale pour la détection de comportements de haut niveau dans des programmes. Ces comportements décrivent des combinaisons de fonctionnalités et sont définis par des formules de logique temporelle du premier ordre. Des behavior patterns, exprimant les réalisations concrètes des fonctionnalités, sont aussi définis par des formules de logique temporelle du premier ordre. L'abstraction de ces fonctionnalités dans des traces de programme est réalisée par réécriture de termes. La validation des traces abstraites par rapport à un comportement de haut niveau donné est réalisée en utilisant les techniques de model checking usuelles. Pour contourner l'indécidabilité générale du problème de construire l'ensemble des formes normales de traces d'un programme donné, on a identifié une propriété des comportements de haut niveau en pratique qui permet d'éviter d'avoir à calculer l'ensemble des formes normales et qui permet d'obtenir un algorithme de détection de complexité linéaire en temps.

Comme pour l'approche basée sur des mots, l'abstraction est une notion clé de notre approche. Travailler sur une forme abstraite des traces de programme et des comportements nous permet d'être indépendants de l'implantation du programme et de traiter des comportements similaires de façon générique, ce qui rend notre approche robuste par rapport aux variantes de codes malicieux.

Chapitre 6

Abstraction Pondérée de Comportements

Dans ce papier, nous étendons notre formalisme d'abstraction afin d'abstraire des traces ne réalisant pas une fonctionnalité de façon certaine. Nous souhaitons prendre en compte d'une part les limitations de l'analyse statique qui entravent la construction du flux de données du programme par des incomplétudes ou des erreurs. Par exemple, supposons que nous souhaitions identifier une écriture de fichier système et qu'une trace du programme analysé représente une ouverture de fichier système suivie d'une écriture d'un fichier apparemment différent du fichier ouvert : plutôt que d'ignorer ce comportement, on souhaite l'interpréter malgré tout comme une écriture de fichier système, mais en tenant compte de l'incertitude due à une éventuelle erreur d'analyse statique. D'autre part, une trace d'exécution peut ne pas être suffisamment expressive et induire un doute quant à l'identification d'une fonctionnalité. Par exemple, si la trace appelle la fonction `sprintf` sur une chaîne de caractères, la chaîne de caractères n'est pas nécessairement entièrement invalidée puisque, en C, la chaîne de caractères est référencée par l'adresse d'un de ses caractères. Enfin, lorsqu'une fonctionnalité est décrite par une séquence complexe d'actions et que cette séquence est observée intégralement exception faite de quelques actions isolées, on peut attribuer cette absence à une erreur d'analyse statique ou au fait que la définition de cette fonctionnalité est incomplète.

Ainsi, lorsqu'une abstraction est incertaine et risque d'induire une erreur d'identification d'un comportement de haut niveau, notre but est de fournir une alternative à l'abstraction classique en attribuant une certaine probabilité à cette abstraction. Aussi, lorsqu'un comportement de haut niveau est recherché dans un programme, on cherche une trace dont une forme abstraite

contienne ce comportement, avec une probabilité dépassant un seuil fixé. Ainsi la détection d'un comportement malicieux peut alors être réalisée de façon plus fine par un moteur antiviral ou par un analyste humain. Les probabilités que le programme exhibe certains comportements peuvent alors venir compléter d'autres caractéristiques du programme calculées par des techniques alternatives d'analyse statique ou dynamique, par exemple des caractéristiques de packing, de profil statistique du code ou des interactions réseaux.

Pour ce faire, nous présentons un formalisme reposant sur un mécanisme de réécriture de termes pondéré, où un poids représente la probabilité que l'abstraction effectuée soit juste. La recherche d'un comportement de haut niveau donné se faisant au niveau de l'ensemble des traces complètement abstraites d'un programme, il faut d'abord construire cet ensemble avant de rechercher le comportement. Cependant un tel ensemble est incalculable dans le cas général. Aussi nous proposons une méthode permettant de détecter un comportement de façon correcte et complète tout en ne construisant que partiellement cet ensemble, pour des comportements ayant une forme particulière et qui s'avère non contraignante en pratique. Par ailleurs, cette méthode fonctionne en temps linéaire, ce qui la rend applicable.

Bien que nous nous intéressions à l'analyse comportementale d'un programme dans un cadre d'analyse statique, c'est-à-dire indépendamment de toute exécution, une application de notre formalisme à une trace capturée par analyse dynamique présente des avantages similaires et est immédiate. En effet, l'ensemble des formes complètement abstraites pour une trace donnée est calculable, et notre technique de détection par abstraction est encore plus facile à utiliser. D'autre part, l'approximation par construction partielle de l'ensemble des formes complètement abstraites, bien qu'elle ne soit plus nécessaire dans ce cas, peut être utilisée à des fins d'optimisation.

6.1 Définitions

Ensembles et Transformations Pondérées

Ensembles pondérés

Un ensemble pondéré sur un semi-anneau $\left(S, \oplus, \otimes, \overline{0}, \overline{1}\right)$ est une application s_A de $A \to S$ où A est un ensemble.

Le support d'un ensemble pondéré $s_A : A \to S$ est noté $Supp\left(s_A\right)$ et défini par : $Supp\left(s_A\right) = \left\{a \in A \,|\, s_A\left(a\right) \neq \overline{0}\right\}$.

L'union de deux ensembles pondérés s_A et s_B sur $\left(S, \oplus, \otimes, \overline{0}, \overline{1}\right)$ est l'ensemble pondéré noté $s_A \cup s_B$ et défini par l'application $s : A \cup B \to S$ telle

que : $\forall c \in A \cup B,\ s(c) = s_A(c) \oplus s_B(c)$.

Transformations pondérées

Une transformation pondérée d'un ensemble A dans un ensemble B sur un semi-anneau $\left(S, \oplus, \otimes, \overline{0}, \overline{1}\right)$ est un ensemble pondéré $s_{A\times B}$ sur S, qui est notée :

$$a \overset{w}{\rightsquigarrow}_{s_{A\times B}} b$$

pour tous $a \in A$, $b \in B$ et $w \in S$ tels que $w = s_{A\times B}(a, b)$. L'indice $s_{A\times B}$ pourra être omis quand il n'y a pas d'ambiguité.

L'union de deux transformations pondérées μ et μ' de $A \times B \to S$, notée $\mu \cup \mu'$, est la transformation pondérée définie, pour tous $a \in A$ et $b \in B$, par :

$$\left(\mu \cup \mu'\right)(a, b) = \mu(a, b) \oplus \mu'(a, b).$$

La composition fonctionnelle de deux transformations pondérées $\mu \in A \times B \to S$ et $\mu' \in B \times C \to S$ est la transformation pondérée $\mu; \mu' \in A \times C \to S$ définie, pour tous $a \in A$ et $c \in C$, par :

$$\left(\mu; \mu'\right)(a, c) = \bigoplus_{b \in B} \mu(a, b) \otimes \mu'(b, c).$$

Soient une transformation pondérée μ d'un ensemble A dans un ensemble B, et un ensemble $L \subseteq A$. On note alors $\mu(L)$ l'ensemble pondéré de $B \to S$ tel que :

$$\forall b \in B,\ \mu(L)(b) = \bigoplus_{a \in L, a \overset{w}{\rightsquigarrow}_{\mu} b} w.$$

On définit de même, pour toute transformation pondérée μ de A dans B et tout ensemble pondéré $s : A \to S$, l'ensemble pondéré $\mu(s) : B \to S$ par :

$$\forall b \in B,\ \mu(s)(b) = \bigoplus_{a \in A, a \overset{w}{\rightsquigarrow}_{\mu} b} s(a) \otimes w.$$

Soit un ensemble pondéré $s : A \to S$. L'identité sur s est la transformation pondérée de A dans A sur S notée Id_s et définie par :

$$\forall a \in A,\ a \overset{s(a)}{\rightsquigarrow}_{Id_s} a.$$

Automates d'Arbres Pondérés

Les automates et transducteurs pondérés (de mots ou d'arbres) ont été étudiés de façon intensive dans la littérature [77, 21, 91, 100, 41], et ont été en particulier appliqués au traitement automatique du langage naturel [86, 73]. L'opération d'addition $\oplus$ du semi-anneau $\left(S, \oplus, \otimes, \overline{0}, \overline{1}\right)$ permet de cumuler les poids de chemins alternatifs dans un automate pondéré, tandis que l'opération de multiplication $\otimes$ permet de cumuler les poids des transitions le long d'un chemin.

Notre définition des automates d'arbres pondérés est une traduction directe en termes d'automates d'arbres au sens de [34] des grammaires d'arbres pondérées définies par Alexandrakis [10] et Knight, May et Vogler [83].

X est un ensemble de variables.

Définition 74 (Automate d'Arbres Pondéré [10])**.** Un *automate d'arbres pondéré* (descendant) sur un semi-anneau $\left(S, \oplus, \otimes, \overline{0}, \overline{1}\right)$ est un quadruplet $A = (\mathcal{F}, Q, q_0, \Delta)$ où $\mathcal{F}$ est un alphabet fini, Q est un ensemble fini d'états, $q_0 \in Q$ est un état initial et Δ est un ensemble fini de règles de la forme :

$$q\left(f\left(x_1, \ldots, x_n\right)\right) \xrightarrow{w} f\left(q_1\left(x_1\right), \ldots, q_n\left(x_n\right)\right)$$

où $f \in \mathcal{F}$ est un symbole d'arité $n \in \mathbb{N}$, $q, q_1, \ldots, q_n \in Q$, $x_1, \ldots, x_n$ sont des variables distinctes d'un ensemble X de variables et $w \in S$.

La relation de transition $\rightarrow_A$ associée à A est définie par :

$$\begin{gathered}
\forall t, t' \in T\left(\mathcal{F} \cup Q\right), \\
t \xrightarrow{w}_A t' \\
\Leftrightarrow \\
\exists\, q(f\left(x_1, \ldots, x_n\right)) \xrightarrow{w} f\left(q_1\left(x_1\right), \ldots, q_n\left(x_n\right)\right) \in \Delta, \\
\exists p \in Pos\left(t\right), \exists u_1, \ldots, u_n \in T\left(\mathcal{F}\right), \\
t\big|_p = q\left(f\left(u_1, \ldots, u_n\right)\right) \\
\text{et } t' = t\left[f\left(q_1\left(u_1\right), \ldots, q_n\left(u_n\right)\right)\right]_p.
\end{gathered}$$

Le poids d'une séquence de réductions par $\rightarrow_A$ est défini comme le produit (par $\otimes$) des poids de chaque réduction. La restriction de $\rightarrow_A$ aux réductions les plus à gauche (*leftmost*) est notée $\rightarrow_A^l$. On note que, dans un semi-anneau commutatif :

$$\forall t, t' \in T\left(\mathcal{F} \cup Q\right), \forall w \in S,\ t \xrightarrow{w}{}_A^{*}\ t' \Leftrightarrow t \xrightarrow{w}{}_A^{l*}\ t'.$$

Le langage d'arbres pondéré reconnu par A est alors l'ensemble pondéré $\|A\| : T\left(\mathcal{F}\right) \rightarrow S$ tel que, pour tout terme $t \in T\left(\mathcal{F}\right)$, $\|A\|\left(t\right)$ soit la somme

des poids de l'ensemble $Red_l(t)$ des séquences de réduction par $\rightarrow_A^l$ de $q_0(t)$ vers t :

$$\forall t \in T(\mathcal{F}),\ \|A\|(t) = \bigoplus_{q_0(t) \overset{w_1}{\rightarrow}{}_A^l \ldots \overset{w_n}{\rightarrow}{}_A^l t \in Red_l(t)} w_1 \otimes \cdots \otimes w_n.$$

Les automates d'arbres pondérés reconnaissent l'ensemble des langages d'arbres réguliers pondérés.

La taille de A est définie par : $|A| = |Q| + |\Delta|$.

Un automate d'arbres non pondéré peut être vu comme un automate d'arbres pondéré A dont toutes les règles ont un poids $\overline{1}$. L'ensemble non pondéré reconnu par A est alors défini comme le support de l'ensemble pondéré reconnu par A. Les automates d'arbres non pondérés reconnaissent l'ensemble des langages d'arbres réguliers non pondérés.

Définition 75. Un *langage de traces pondéré* sur un alphabet $\mathcal{F}$ est un langage d'arbres pondéré dans $T_{Trace}(\mathcal{F})$. Un *automate de traces pondéré* sur $\mathcal{F}$ est un automate d'arbres pondéré reconnaissant un langage de traces pondéré.

Transducteurs d'Arbres Pondérés

Définition 76 (Transducteur d'Arbres Pondéré [83])**.** Un *transducteur d'arbres pondéré* (descendant) sur un semi-anneau $\left(S, \oplus, \otimes, \overline{0}, \overline{1}\right)$ est un quintuplet $\tau = (\mathcal{F}, \mathcal{F}', Q, q_0, \Delta)$ où $\mathcal{F}$ est un ensemble fini de symboles d'entrée, $\mathcal{F}'$ est un ensemble fini de symboles de sortie, Q est un ensemble fini d'états, $q_0 \in Q$ est un état initial et Δ est un ensemble fini de règles de la forme :

$$\begin{array}{ll} q(f(x_1, \ldots, x_n)) \overset{w}{\rightarrow} u & \\ \text{ou} & \\ q(x_1) \overset{w}{\rightarrow} u & (\epsilon\text{-transition}) \end{array}$$

où $f \in \mathcal{F}$ est un symbole d'arité $n \in \mathbb{N}$, $q \in Q$, $x_1, \ldots, x_n$ sont des variables distinctes d'un ensemble X de variables, $u \in T(\mathcal{F}' \cup Q, \{x_1, \ldots, x_n\})$ et $w \in S$.

La relation de transition $\rightarrow_\tau$ associée au transducteur τ est définie par :

$$\begin{array}{c} \forall t, t' \in T(\mathcal{F} \cup \mathcal{F}' \cup Q), \\ t \overset{w}{\rightarrow}_\tau t' \\ \Leftrightarrow \\ \exists\, q(f(x_1, \ldots, x_n)) \overset{w}{\rightarrow} u \in \Delta, \\ \exists p \in Pos(t), \exists u_1, \ldots, u_n \in T(\mathcal{F}'), \\ t|_p = q(f(u_1, \ldots, u_n)) \\ \text{et } t' = t\left[u\{x_1 \leftarrow u_1, \ldots, x_n \leftarrow u_n\}\right]_p. \end{array}$$

Les ϵ-transitions sont un cas particulier de cette définition.

Le poids d'une séquence de réductions par $\rightarrow_\tau$ est défini comme le produit (par $\otimes$) des poids de chaque réduction. La restriction de $\rightarrow_\tau$ aux réductions les plus à gauche (*leftmost*) est notée $\rightarrow_\tau^l$.

La transformation pondérée réalisée par τ est alors la transformation pondérée $\|\tau\|$ de $T(\mathcal{F})$ dans $T(\mathcal{F}')$ sur S telle que, pour tous termes $t \in T(\mathcal{F})$ et $t' \in T(\mathcal{F}')$, $\|\tau\|(t,t')$ est égal à la somme des poids de l'ensemble $Red_l(t)$ des séquences de réduction par $\rightarrow_\tau^l$ de $q_0(t)$ vers t' :

$$\forall t \in T(\mathcal{F}), \forall t' \in T(\mathcal{F}'),$$
$$\|\tau\|(t,t') = \bigoplus_{q_0(t) \overset{w_1}{\rightarrow}{}_\tau^l \cdots \overset{w_n}{\rightarrow}{}_\tau^l t' \in Red_l(t)} w_1 \otimes \cdots \otimes w_n.$$

Un transducteur d'arbres pondéré descendant est *linéaire* si aucune variable n'apparaît deux fois dans un membre gauche ou droit de règle. Il est *non effaçant* si, pour chaque règle $q(f(x_1,\ldots,x_n)) \overset{w}{\rightarrow} u$ et pour tout $i \in [1..n]$, la variable x_i apparaît dans u.

Une transformation pondérée de $T(\mathcal{F})$ dans $T(\mathcal{F}')$ sur S est dite *rationnelle* ssi il existe un transducteur d'arbres pondéré descendant linéaire non effaçant qui la réalise.

Les transducteurs d'arbres pondérés descendants linéaires non effaçants préservent la régularité [83, 76] et sont fermés par union et composition fonctionnelle [83, 44]. Dans la suite, nous ne considérerons que des transducteurs d'arbres pondérés descendants linéaires non effaçants, en raison de leurs bonnes propriétés.

Un transducteur d'arbres non pondéré peut être vu comme un transducteur d'arbres pondéré τ dont toutes les règles ont un poids $\overline{1}$. La transformation pondérée $\|\tau\|$ reconnue par τ peut alors être confondue avec son support.

La taille de τ est définie par : $|\tau| = |Q| + |\Delta|$.

Définition 77. Un *transducteur de traces pondéré* est un transducteur pondéré transformant des langages de traces pondérés en des langages de traces pondérés.

Définition 78 (Relabeling)**.** Soit deux alphabets $\mathcal{F}$ et $\mathcal{F}'$ pas nécessairement distincts. Un *relabeling* de $\mathcal{F}$ dans $\mathcal{F}'$ est un homomorphisme d'arbres de $T(\mathcal{F},X) \rightarrow T(\mathcal{F}',X)$ défini de façon unique par une application totale de $\mathcal{F} \rightarrow \mathcal{F}'$.

Proposition 79. *Soit deux alphabets $\mathcal{F}$ et $\mathcal{F}'$. Tout relabeling de $\mathcal{F}$ dans $\mathcal{F}'$ est réalisé par un transducteur d'arbres descendant $(\mathcal{F},\mathcal{F}',Q,q_0,\Delta)$ tel que Q contient un unique état $*$ et les règles de Δ sont de la forme : $*(f(x_1,\ldots,x_k)) \rightarrow g(*(x_1),\ldots,*(x_k))$ avec $k \in \mathbb{N}$, $f \in \mathcal{F}$ et $g \in \mathcal{F}'$ d'arité k.*

Démonstration. Ce transducteur doit simplement transformer tout symbole de $\mathcal{F}$ en son symbole associé sur $\mathcal{F}'$. □

Résultats de Complexité

Proposition 80. *Soit un automate d'arbres pondéré A sur un alphabet $\mathcal{F}$. Alors l'identité sur $\|A\|$ est réalisée par un transducteur d'arbres pondéré descendant linéaire non effaçant sans ϵ-transitions de taille $O(|A|)$.*

Démonstration. Immédiat car un automate d'arbres pondéré a déjà une forme de transducteur d'arbres pondéré réalisant précisément l'identité sur le langage d'arbres pondéré reconnu par l'automate. □

Proposition 81. *Soit un automate de traces pondéré A sur un alphabet $\mathcal{F}$ et un semi-anneau $\left(S, \oplus, \otimes, \overline{0}, \overline{1}\right)$ et soit un transducteur de traces pondéré $\tau = (\mathcal{F}, \mathcal{F}', Q, q_0, \Delta)$. Alors le langage de traces pondéré $\tau(\|A\|)$ est reconnu par un automate de traces pondéré de taille $O(|\tau| \cdot |A|)$.*

Démonstration. Définissons deux alphabets de mots Ω et Ω' en bijection avec les ensembles finis $T_{Action}(\mathcal{F})$ et $T_{Action}(\mathcal{F}')$. Cela induit une bijection entre les mots de Ω^* (resp. Ω'^*) et les traces de $T_{Trace}(\mathcal{F})$ (resp. $T_{Trace}(\mathcal{F}')$).

Alors, A étant un automate de traces pondéré et τ un transducteur de traces pondéré, le résultat découle par analogie directe avec le cas des transducteurs pondérés de mots sur les alphabets de mots Ω et Ω' : le résultat équivalent sur les mots est montré notamment dans [87].

□

6.2 Abstraction Pondérée

Nous définissons l'abstraction pondérée comme une extension du formalisme d'abstraction non pondéré. La notion de behavior pattern reste inchangée, un behavior pattern étant défini comme l'ensemble des traces réalisant une certaine fonctionnalité.

Définition 82. Un *behavior pattern* est un ensemble de traces $B \subseteq T_{Trace}(\mathcal{F}_\Sigma)$ validant une formule FOLTL close φ sur $AP = \Sigma$:

$$B = \{t \in T_{Trace}(\mathcal{F}_\Sigma) \mid t \models \varphi\}.$$

Transformation d'Abstraction Pondérée

De même, l'abstraction d'un behavior pattern dans une trace consiste toujours à identifier une occurrence de ce behavior pattern dans la trace et à marquer cette occurrence en insérant une action abstraite à son niveau. Cette action abstraite est de la forme $\lambda(d_1, \ldots, d_n)$, où λ est le symbole de Γ associé au behavior pattern et $d_1, \ldots, d_n$ sont des constantes de $\mathcal{F}_d$. La différence avec le cas non pondéré est que désormais certaines occurrences du behavior pattern peuvent réaliser la fonctionnalité avec une certaine incertitude. Une forme abstraite d'une trace d'un programme a donc un degré d'incertitude associé et le problème de la détection est alors de déterminer s'il existe une forme abstraite réalisant un comportement abstrait donné avec un degré d'incertitude acceptable.

De façon naturelle, l'incertitude de l'abstraction est décrite par la probabilité qu'une occurrence du behavior pattern réalise effectivement la fonctionnalité.

Exemple 83. Définissons trois behavior patterns $\lambda_1 := a$, $\lambda_2 := b$ et $\lambda_3 := c$, tels que la trace b ne réalise la fonctionnalité associée au behavior pattern λ_2 qu'avec une probabilité 0.1.

Supposons alors que nous souhaitions déterminer si un programme donné exhibe, avec une probabilité supérieure à 0.5, le comportement abstrait décrit par la formule FOLTL : $\lambda_1 \wedge \neg\lambda_2 \,\mathbf{U}\, \lambda_3$. Une façon de procéder est de calculer l'ensemble des formes complètement abstraites des traces du programme puis de chercher si l'une de ces formes abstraites est une instance de ce comportement et a une probabilité d'abstraction supérieure au seuil de 0.5.

Ainsi, considérons un programme dont l'unique trace est $a \cdot b \cdot c$. L'ensemble de ses formes complètement abstraites est composé de deux éléments, la trace $a \cdot \lambda_1 \cdot b \cdot \lambda_2 \cdot c \cdot \lambda_3$ qui a une probabilité d'abstraction de 0.1 et la trace $a \cdot \lambda_1 \cdot b \cdot c \cdot \lambda_3$ qui a une probabilité d'abstraction de 0.9, et l'on en déduit immédiatement que le programme exhibe bien le comportement recherché.

Bien que l'incertitude de l'abstraction soit représentée intuitivement par une probabilité, nous formalisons la relation d'abstraction résultante à l'aide d'un système de réécriture non pas probabiliste mais pondéré. En effet, une mesure de probabilité est trop contraignante dans notre formalisme. Définissons par exemple un behavior pattern $\lambda := a \cdot c$ et un behavior pattern $\lambda' := b \cdot c$ abstraits tous deux avec probabilité 1. La trace $a \cdot b \cdot c$ s'abstrait alors en $a \cdot b \cdot c \cdot \lambda \cdot \lambda'$ avec une certaine probabilité p et en $a \cdot b \cdot c \cdot \lambda' \cdot \lambda$ avec probabilité $1 - p$. Pour peu que nous cherchions le comportement $\lambda \cdot \lambda'$ avec une probabilité strictement supérieure à p, la détection échouerait, à tort. La raison en

est que nous nous intéressons non pas à la probabilité d'une forme abstraite relativement aux autres formes abstraites mais à la probabilité de la chaîne d'abstractions permettant d'obtenir cette forme abstraite.

Pour cette raison, nous représentons l'incertitude de l'abstraction par un poids défini sur un semi-anneau commutatif particulier, le semi-anneau tropical $(\mathbb{R}^+ \cup \{+\infty\}, \min, +, +\infty, 0)$, dans lequel un poids w est naturellement lié à une probabilité p par la formule : $w = -\log(p)$. Ainsi, si une première abstraction a une probabilité p_1, elle est associée au poids $-\log(p_1)$, et si une seconde abstraction a une probabilité p_2, elle est associée au poids $-\log(p_2)$, et le cumul de ces deux poids $w_1 \otimes w_2 = w_1 + w_2 = -\log(p_1 \cdot p_2)$ représente bien la probabilité cumulée de réaliser ces deux abstractions. Par ailleurs, si une forme abstraite peut être obtenue par deux chaînes d'abstractions différentes, avec des poids différents, seule nous intéresse la chaîne d'abstraction avec la plus faible incertitude, puisqu'in fine il s'agit de découvrir une forme abstraite dont le poids ne dépasse pas un seuil donné : le choix de min comme opérateur de cumul des poids d'une même forme abstraite est donc naturel, étant donné que le poids croît en fonction inverse de la probabilité. Enfin, le semi-anneau tropical permet d'obtenir les résultats de décidabilité de la détection en Section 6.3.

Notons que l'abstraction pondérée pourrait être définie de façon générique par rapport au semi-anneau commutatif choisi pour représenter les poids. Pour cette raison et par souci de simplicité, nous conservons la notation $\left(S, \oplus, \otimes, \overline{0}, \overline{1}\right)$ pour représenter le semi-anneau tropical.

On formalise alors l'abstraction pondérée à l'aide d'un système de réécriture pondéré, que nous appelons système d'abstraction pondéré.

Comme dans le formalisme d'abstraction non pondérée, nous insérons une action abstraite de $T_{Action}(\mathcal{F}_\Gamma)$ lorsqu'une occurrence d'un behavior pattern est découverte. Ainsi, la trace $a \cdot b \cdot c$ pourra par exemple être réécrite en $a \cdot \lambda_1 \cdot b \cdot c$.

Lorsqu'une occurrence d'un behavior pattern n'est pas reconnue de façon certaine, nous l'abstrayons avec une probabilité p, mais nous devons alors aussi considérer le cas complémentaire avec la probabilité complémentaire pour le pattern de ne pas être reconnu. Ainsi, pour une occurrence d'un behavior pattern abstraite en λ avec une probabilité p, nous réalisons une abstraction complémentaire de cette occurrence en un symbole dual de λ dans Γ, noté $\overline{\lambda}$, avec probabilité $1 - p$. L'exemple précédent permet de juger de l'importance d'une telle abstraction complémentaire. En effet, si on avait choisi de simplement réécrire $a \cdot b \cdot c$ en $a \cdot b \cdot \lambda_2 \cdot c$ avec une probabilité 0.1, la trace $a \cdot b \cdot c$ n'admettrait alors qu'une seule forme normale, $a \cdot \lambda_1 \cdot b \cdot \lambda_2 \cdot c \cdot \lambda_3$ de probabilité

0.1, et l'on en déduirait à tort que le programme n'exhibe pas le comportement $\lambda_1 \wedge \neg\lambda_2 \,\mathbf{U}\, \lambda_3$. Au lieu de cela, la trace, la trace $a \cdot b \cdot c$ peut également être réécrite en $a \cdot b \cdot \overline{\lambda_2} \cdot c$ avec une probabilité 0.9, où $\overline{\lambda_2}$ est un symbole de Γ associé à λ_2 de façon unique, ce qui produit la forme normale $a \cdot \lambda_1 \cdot b \cdot \overline{\lambda_2} \cdot c \cdot \lambda_3$, de probabilité 0.9.

Ainsi Γ contient les symboles d'abstraction et leurs duaux.

Le système d'abstraction pondéré considéré en exemple est donc composé des trois règles suivantes :

$$\begin{array}{lcl} a & \rightarrow & \left\{a \cdot \lambda_1 : \overline{1}\right\} \\ b & \rightarrow & \left\{b \cdot \lambda_2 : 0.1, b \cdot \overline{\lambda_2} : 0.9\right\} \\ c & \rightarrow & \left\{c \cdot \lambda_1 : \overline{1}\right\}. \end{array}$$

Lorsqu'il n'y a aucune ambiguité, comme pour le système précédent, nous indiquerons à la place des poids w la probabilité p telle que : $w = -\log(p)$.

Définition 84 (Système d'Abstraction Pondéré)**.** Soit $\lambda \in \Gamma$ un symbole d'abstraction et $\overline{\lambda} \in \Gamma$ son symbole dual, X un ensemble de variables de sorte *Data*, $\overline{x}$ une séquence de variables de X et y une variable de sorte *Trace*. Un *système d'abstraction pondéré* R sur $T_{Trace}(\mathcal{F}, X \cup \{y\})$, sur un semi-anneau $\left(S, \oplus, \otimes, \overline{0}, \overline{1}\right)$, est un ensemble fini de règles de réécriture de la forme :

$$\begin{array}{rl} A_i(X) \cdot B_i(X) \cdot y \rightarrow & \{\, A_i(X) \cdot \lambda(\overline{x}) \cdot B_i(X) \cdot y : w_i, \\ & \quad A_i(X) \cdot \overline{\lambda}(\overline{x}) \cdot B_i(X) \cdot y : w'_i \,\} \end{array}$$

ou de la forme :

$$A_i(X) \cdot B_i(X) \cdot y \rightarrow A_i(X) \cdot \lambda(\overline{x}) \cdot B_i(X) \cdot y : \overline{1}$$

où les ensembles $A_i(X)$ et $B_i(X)$ sont des ensembles de traces concrètes de $T_{Trace}(\mathcal{F}_\Sigma, X)$, $w_i, w'_i \in S$.

Le système d'abstraction non pondéré associé est le système R_u composé des mêmes règles, où les poids ne sont pas considérés.

Notons que, le poids représentant la probabilité de l'abstraction d'une occurrence du behavior pattern, il doit exister pour tout i une probabilité $p_i \in [0..1]$ telle que : $w_i = -\log(p_i)$ et $w'_i = -\log(1 - p_i) = -\log(1 - e^{-w_i})$.

Définition 85. La *relation de réduction pondérée* sur $T_{Trace}(\mathcal{F})$ générée par un système d'abstraction pondéré R sur un semi-anneau $\left(S, \oplus, \otimes, \overline{0}, \overline{1}\right)$ avec n règles de réécriture $A_i(X) \cdot B_i(X) \cdot y \rightarrow \{A_i(X) \cdot \lambda(\overline{x}) \cdot B_i(X) \cdot y : w_i, A_i(X) \cdot$

$\overline{\lambda}(\overline{x}) \cdot B_i(X) \cdot y : w'_i\}$ est la relation, également notée R, telle que, pour tous $t, t' \in T_{Trace}(\mathcal{F})$ et pour tout $w \in S$:

$$t \xrightarrow{w}_R t'$$
$$\Leftrightarrow$$
$$\exists \sigma \in Inst_{X \cup \{y\}}, \exists p \in Pos(t), \exists i \in [1..n], \exists \mu \in \Gamma, (\mu, w) \in \left\{(\lambda, w_i), \left(\overline{\lambda}, w'_i\right)\right\},$$
$$\exists a \in T_{Trace}(\mathcal{F}) \cdot T_{Action}(\mathcal{F}_\Sigma), \exists b \in T_{Trace}(\mathcal{F}),$$
$$a|_\Sigma \in A_i(X)\sigma, b|_\Sigma \in B_i(X)\sigma,$$
$$t|_p = a \cdot b \cdot y\sigma \text{ et } t' = t[a \cdot \mu(\overline{x})\sigma \cdot b \cdot y\sigma]_p.$$

La relation de réduction non pondérée associée est la relation de réduction induite par le système d'abstraction non pondéré R_u et est également notée R_u.

La transformation générée par un système d'abstraction pondéré est une transformation pondérée telle qu'un terme $t \in T_{Trace}(\mathcal{F})$ soit transformé en un terme $t' \in T_{Trace}(\mathcal{F})$ avec un poids w correspondant à la somme des poids de l'ensemble des pas de réécriture par lesquels t peut se réécrire en t'.

Définition 86. La *transformation pondérée* sur $T_{Trace}(\mathcal{F})$ générée par un système d'abstraction pondéré R sur un semi-anneau $\left(S, \oplus, \otimes, \overline{0}, \overline{1}\right)$ avec n règles de réécriture $A_i(X) \cdot B_i(X) \cdot y \to \{A_i(X) \cdot \lambda(\overline{x}) \cdot B_i(X) \cdot y : w_i, A_i(X) \cdot \overline{\lambda}(\overline{x}) \cdot B_i(X) \cdot y : w'_i\}$ est la transformation pondérée définie comme suit :

$$\forall t, t' \in T_{Trace}(\mathcal{F}),$$
$$t \overset{w}{\rightsquigarrow}_R t'$$
$$\Leftrightarrow$$
$$\text{il existe } n \text{ pas de réduction pondérée } t \xrightarrow{w_i}_R t' \text{ et } w = \bigoplus_{i \in [1..n]} w_i.$$

Une abstraction pondérée par rapport à un behavior pattern donné est alors la transformation pondérée générée sur $T_{Trace}(\mathcal{F})$ par un système d'abstraction pondéré, telle que l'ensemble des instances des membres gauches du système d'abstraction couvre l'ensemble des traces du behavior pattern.

Définition 87 (Transformation d'Abstraction Pondérée)**.** Soit B un behavior pattern associé à un symbole d'abstraction $\lambda \in \Gamma$. Soit X un ensemble de variables de sorte *Data* et y une variable de sorte *Trace*. La *transformation d'abstraction pondérée* par rapport à ce behavior pattern sur le semi-anneau $\left(S, \oplus, \otimes, \overline{0}, \overline{1}\right)$ est la transformation pondérée sur $T_{Trace}(\mathcal{F})$ générée par le système d'abstraction pondéré sur $\left(S, \oplus, \otimes, \overline{0}, \overline{1}\right)$ composé de n règles

$A_i(X) \cdot B_i(X) \cdot y \to \{ A_i(X) \cdot \lambda(\overline{x}) \cdot B_i(X) \cdot y : w_i,\ A_i(X) \cdot \overline{\lambda}(\overline{x}) \cdot B_i(X) \cdot y : w'_i \}$ vérifiant :

$$B = \bigcup_{i \in [1..n]} \bigcup_{\sigma \in Inst_X} (A_i(X) \cdot B_i(X))\, \sigma.$$

Puis on généralise la définition d'une transformation d'abstraction pondérée à un ensemble de behavior patterns.

Définition 88. Soit C un ensemble fini de behavior patterns. La *transformation d'abstraction pondérée* par rapport à C sur un semi-anneau $\left(S, \oplus, \otimes, \overline{0}, \overline{1}\right)$ est l'union des transformations d'abstraction par rapport à chaque behavior pattern de C sur le semi-anneau $\left(S, \oplus, \otimes, \overline{0}, \overline{1}\right)$.

Comme dans le chapitre précédent, on décrit par la notation $\lambda := \varphi$ un behavior pattern défini par une formule FOLTL φ et associé à un symbole d'abstraction λ.

Abstraction Pondérée Saine

Il n'y aurait pas de sens à abstraire une occurrence donnée d'un behavior pattern à la fois en un symbole abstrait et en son dual. De même, abstraire deux fois la même occurrence d'un behavior pattern n'aurait pas de sens et fausserait le calcul du poids de l'abstraction.

Aussi nous définissons une transformation d'abstraction pondérée saine. Dans cette définition, pour une action abstraite $\alpha \in T_{Action}(\mathcal{F}_\Gamma)$, on note $\overline{\alpha}$ l'action duale construite en remplaçant dans α le symbole de Γ par son dual. Par exemple, pour $\alpha = \lambda(d)$, avec $d \in \mathcal{F}_d$, on définit $\overline{\alpha} = \overline{\lambda}(d)$, et pour $\alpha = \overline{\lambda}(d)$, on définit $\overline{\alpha} = \lambda(d)$.

Définition 89 (Transformation d'Abstraction Pondérée Saine)**.** La *transformation d'abstraction pondérée saine* sur un semi-anneau $\left(S, \oplus, \otimes, \overline{0}, \overline{1}\right)$ une transformation d'abstraction pondérée $\rightsquigarrow_R$ est la transformation pondérée $\rightsquigarrow_{R_s}$ définie par :

$$\begin{gathered}
\forall t_1, t_2 \in T_{Trace}(\mathcal{F}), \forall \alpha \in T_{Action}(\mathcal{F}_\Gamma), \forall w \in S \\
t_1 \cdot t_2 \overset{w}{\rightsquigarrow}_{R_s} t_1 \cdot \alpha \cdot t_2 \\
\Leftrightarrow \\
t_1 \cdot t_2 \overset{w}{\rightsquigarrow}_{R} t_1 \cdot \alpha \cdot t_2 \\
\text{et} \\
\nexists (u, u') \in T_{Trace}(\mathcal{F}_\Gamma) \times T_{Trace}(\mathcal{F}), \\
t_2 = u \cdot \alpha \cdot u' \text{ or } t_2 = u \cdot \overline{\alpha} \cdot u'.
\end{gathered}$$

Observons qu'une transformation d'abstraction pondérée saine est terminante.

Désormais, pour un système d'abstraction pondéré R, on note $R_{\rightsquigarrow}$ la transformation d'abstraction pondérée saine qu'il génère sur $T_{Trace}(\mathcal{F})$. On définit alors la transformation pondérée $R^*_{\rightsquigarrow}$ par : $R^*_{\rightsquigarrow} = \bigcup_{i\in\mathbb{N}} R^i_{\rightsquigarrow}$.

6.3 Problème de la détection

Dans la suite, nous supposons que $R_{\rightsquigarrow}$ est une transformation d'abstraction pondérée saine sur le semi-anneau $\left(S, \oplus, \otimes, \overline{0}, \overline{1}\right)$.

Comme dans le cas non pondéré, un comportement abstrait décrit des combinaisons de fonctionnalités de haut niveau, c'est-à-dire des séquences d'actions abstraites, et est défini à l'aide d'une formule FOLTL sur $AP = \Gamma$.

Définition 90. Un *comportement abstrait* est un ensemble de traces $M \subseteq T_{Trace}(\mathcal{F}_\Gamma)$ validant une formule FOLTL close φ_M sur $AP = \Gamma$:

$$M = \{t \in T_{Trace}(\mathcal{F}_\Gamma) \mid t \models \varphi_M\}.$$

De même, comme dans le chapitre précédent, l'opérateur initial $\mathbf{F}$ sera implicite dans les définitions des comportements abstraits et un comportement M défini par une formule φ_M sera décrit par la notation : $M := \varphi_M$.

Dans notre formalisme pondéré, la détection d'un comportement abstrait est définie par rapport à un seuil $\rho \in S \setminus \left\{\overline{0}\right\}$ et un programme p exhibe un comportement abstrait M si une forme abstraite d'une de ses traces réalise M avec un poids ne dépassant pas ce seuil ρ. Mais plutôt que de chercher cette forme abstraite parmi les formes normales de traces de p, comme dans le cas non pondéré, on la cherche parmi les formes partiellement abstraites de traces de p et on requiert alors que ses descendants restent infectés, quelque soit leur poids. En effet, travailler sur les formes normales implique que toute occurrence d'un behavior pattern est bien abstraite, même les occurrences ne jouant aucun rôle dans la détection du comportement abstrait. Or, toute abstraction est susceptible de modifier le poids final et donc de compromettre la détection en dépassant le seuil ρ. Remarquons par ailleurs que, dans le cas non pondéré, ces deux définitions sont équivalentes.

Exemple 91. Définissons le comportement $M := \lambda_1 \wedge \neg\lambda_2 \,\mathbf{U}\, \lambda_3$. Détecter M revient alors à trouver une trace partiellement abstraite réalisant M donc contenant les actions λ_1 et λ_3 et dont les occurrences du behavior pattern λ_2 apparaissant entre les actions λ_1 et λ_3 ont déjà été abstraites en $\overline{\lambda_2}$.

L'interprétation du seuil par rapport auquel la détection est définie dépend du semi-anneau choisi. On note donc $\preceq$ la relation d'ordre à vérifier : dans le semi-anneau tropical, on aura $\preceq = \leq$. On choisit d'inclure l'égalité afin de pouvoir considérer comme cas particulier la détection "certaine", i.e. $\rho = \overline{1}$.

Définition 92. Un ensemble de traces L *exhibe un comportement abstrait* M défini par une formule φ_M par rapport à un seuil $\rho \in S \setminus \{\overline{0}\}$, noté $L \Cap_{\preceq \rho} M$, ssi :

$$\begin{gathered}\exists t \in L, \exists t' \in T_{Trace}(\mathcal{F}), \\ t' \models \varphi_M,\ R^*_{\rightsquigarrow}(t,t') \preceq \rho \\ \text{et} \\ \forall t'' \in R^*_u(t'),\ t''|_\Gamma \models \varphi_M.\end{gathered}$$

L'ensemble pondéré $R^*_{\rightsquigarrow}(t,t')$ n'étant pas calculable en général, nous généralisons la propriété de (m,n)-complétude au cas pondéré. Cette propriété exprime le fait que s'il existe une trace partiellement abstraite t' comme dans la Définition 92, alors elle peut être découverte en au plus m étapes d'abstraction et n étapes d'abstraction suffisent à vérifier que l'ensemble de ses descendants réalise toujours M.

Intuitivement, les m étapes permettent de faire apparaître dans une trace exhibant M les actions abstraites garantissant la reconnaissance de M (par exemple les actions λ_1 et λ_3 dans le cas de la signature $\lambda_1 \wedge \neg\lambda_2 \,\mathbf{U}\, \lambda_3$), et les n étapes permettent de vérifier que toutes les abstractions importantes ont été réalisées (par exemple qu'aucune abstraction ne peut insérer une action λ_2 entre les actions λ_1 et λ_3). Ainsi, on peut considérer que le coût (et donc le seuil) de l'abstraction ne porte que sur les m premières étapes.

Définition 93 ((m,n)-complétude)**.** Soit M un comportement abstrait et m et n des entiers positifs. M a la propriété de (m,n)-*complétude* ssi pour tout seuil $\rho \in S \setminus \{\overline{0}\}$ et pour tout ensemble de traces $L \subseteq T_{Trace}(\mathcal{F}_\Sigma)$:

$$\begin{gathered}L \Cap_{\preceq \rho} M \\ \Leftrightarrow \\ \exists t \in L, \exists t' \in T_{Trace}(\mathcal{F}), \exists i \leq m,\ R^i_{\rightsquigarrow}(t,t') \preceq \rho \\ \text{et} \\ \forall t'' \in R^{\leq n}_{\overline{u}}(t'),\ t''|_\Gamma \models \varphi_M.\end{gathered}$$

Exemple 94. Considérons le comportement $M := \lambda_1 \wedge \neg\lambda_2 \,\mathbf{U}\, \lambda_3$ et notons w_2 le poids minimum d'une règle insérant une action $\overline{\lambda_2}$. Les valeurs de m et n sont définies comme suit.

Intuitivement, il faut que m vaille au moins 2 pour faire apparaître les actions λ_1 et λ_3. De plus, si des occurrences du behavior pattern λ_2 existent

entre ces deux actions, il faut les avoir abstraites en $\overline{\lambda_2}$ durant les m étapes (autrement, elles pourront être abstraites en λ_2 durant les n étapes et faire sortir du comportement M). Or si la trace t' existe, elle a un poids ne dépassant pas le seuil ρ donc au plus ρ/w_2 occurrences du behavior pattern λ_2 ont été abstraites en $\overline{\lambda_2}$. On prend donc $m = 2 + \rho/w_2$.

Supposons maintenant que, pour un certain $i \leq m$, nous ayons une telle trace t' dans $R^i_{\rightsquigarrow}(L)$ qui réalise M avec un poids ne dépassant pas le seuil ρ. Pour déterminer si ses descendants réalisent toujours M, il suffit de regarder tous ses descendants à l'ordre 1 pour vérifier qu'aucune occurrence du behavior pattern λ_2 n'a été oubliée entre les actions λ_1 et λ_3. En effet, si une occurrence du behavior pattern λ_2 peut être abstraite au bout de p réductions de t', elle peut être abstraite directement dans t'. Cette propriété de permutation des étapes d'abstraction sera formellement établie dans la suite. On prend donc $n = 1$.

Nous montrerons dans le Théorème 107 que ces valeurs sont correctes. Constatons par ailleurs que si $w_2 = \overline{0}$, i.e. si les occurrences du behavior pattern λ_2 ne sont jamais abstraites en $\overline{\lambda_2}$, on retrouve la $(2,1)$-complétude du cas non pondéré.

Nous montrons maintenant que la détection d'un comportement ayant la propriété de (m,n)-complétude est décidable dans le cas d'une transformation d'abstraction pondérée rationnelle.

Pour cela, on définit l'ensemble, non pondéré, des traces réalisant M et dont les descendants jusqu'à l'ordre n réalisent toujours M.

Définition 95. Soit R une transformation d'abstraction pondérée saine. Soit M un comportement défini par une formule φ_M ayant la propriété de (m,n)-complétude. L'*ensemble des traces n-exhibant M* par rapport à R est l'ensemble :

$$\left\{t \in T_{Trace}(\mathcal{F}) \mid \forall t' \in R_u^{\leq n}(t),\ t'\big|_\Gamma \models \varphi_M\right\}.$$

Lorsque M est régulier et que l'inverse de la relation d'abstraction non pondérée R_u est rationnelle, l'ensembles des traces n-exhibant M est régulier.

Lemme 96. *Soit $R_{\rightsquigarrow}$ une transformation d'abstraction pondérée saine sur un semi-anneau $\left(S, \oplus, \otimes, \overline{0}, \overline{1}\right)$, telle que la relation R_u^{-1} soit rationnelle. Soit M un comportement abstrait régulier ayant la propriété de (m,n)-complétude pour deux entiers positifs m et n.*

Alors l'ensemble de traces n-exhibant M est régulier.

Démonstration. L'ensemble des traces n-exhibant M étant défini par :

$$\left\{t \in T_{Trace}(\mathcal{F}) \mid \forall t' \in R_u^{\leq n}(t),\ t'\big|_\Gamma \models \varphi_M\right\}$$

et R_u étant une relation d'abstraction non pondérée, ce résultat est une instance du Lemme 57. □

La décidabilité de la détection dans le cas d'un comportement ayant la propriété de (m, n)-complétude s'ensuit alors.

Théorème 97. *Soit $R_{\rightsquigarrow}$ une transformation d'abstraction saine sur le semi-anneau tropical $\left(S, \oplus, \otimes, \overline{0}, \overline{1}\right)$ telle que $R_{\rightsquigarrow}$ et R_u^{-1} soient rationnelles. Il existe une procédure de détection décidant si $L \Cap_{\preceq \rho} M$, pour tout ensemble régulier de traces L, pour tout seuil $\rho \in S \setminus \left\{\overline{0}\right\}$ et pour tout comportement abstrait régulier M ayant la propriété de (m, n)-complétude pour deux entiers positifs m et n.*

Démonstration. Définissons, dans le semi-anneau tropical :

$$R_{\rightsquigarrow}^{\leq m} = \bigcup_{0 \leq i \leq m} R_{\rightsquigarrow}^{i}.$$

Alors, pour tout $t' \in T_{Trace}(\mathcal{F})$:

$$R_{\rightsquigarrow}^{\leq m}(L)(t') = \bigoplus_{t \in L} R_{\rightsquigarrow}^{\leq m}(t, t') = \bigoplus_{t \in L} \bigoplus_{0 \leq i \leq m} R_{\rightsquigarrow}^{i}(t, t').$$

En notant que $\oplus = \min$ et $\preceq = \leq$ dans le semi-anneau tropical, la propriété de (m, n)-complétude s'énonce alors ainsi :

$$\begin{gathered} L \Cap_{\preceq \rho} M \\ \Leftrightarrow \\ \exists t' \in T_{Trace}(\mathcal{F}),\ R_{\rightsquigarrow}^{\leq m}(L)(t') \preceq \rho \\ \text{et} \\ \forall t'' \in R_u^{\leq n}(t'),\ t''|_{\Gamma} \models \varphi_M. \end{gathered}$$

En notant M'' l'ensemble des traces n-exhibant M et $Id_{M''}$ l'identité sur M'', la (m, n)-complétude sur M peut être reformulée en :

$$\begin{gathered} L \Cap_{\preceq \rho} M \\ \Leftrightarrow \\ \exists t' \in M'',\ R_{\rightsquigarrow}^{\leq m}(L)(t') \preceq \rho \\ \Leftrightarrow \\ \exists t' \in T_{Trace}(\mathcal{F}),\ Id_{M''}\left(R_{\rightsquigarrow}^{\leq m}(L)\right)(t') \preceq \rho. \end{gathered}$$

M'' est régulier, d'après le Lemme 96 donc $Id_{M''}$ est rationnel, d'après la Proposition 80. De plus, $R_{\rightsquigarrow}$ est rationnelle et L est régulier, donc $R_{\rightsquigarrow}^{\leq m}(L)$ est régulier, de même que $Id_{M''}\left(R_{\rightsquigarrow}^{\leq m}(L)\right)$.

Enfin, dans le semi-anneau tropical, la recherche du chemin de plus petit poids dans un automate d'arbres pondéré est linéaire [62].

□

On montre maintenant que les comportements abstraits considérés en pratique ont la propriété de (m, n)-complétude. Pour cela, on établit plusieurs résultats préliminaires.

Définition 98 (Position Concrète)**.** Soit t un terme de $T_{Trace}(\mathcal{F})$ et t' un sous-terme de t de sorte *Trace*. La *position concrète de* t' dans t est la position de $t'|_\Sigma$ dans $t|_\Sigma$.

Définition 99 (Réduction à une Position Concrète)**.** Soit R un système d'abstraction pondéré sur un semi-anneau $\left(S, \oplus, \otimes, \overline{0}, \overline{1}\right)$. On dit qu'une trace $t = t_1 \cdot t_2$ *est réduite par* R *en* $t_1 \cdot \alpha \cdot t_2$ *à la position concrète* p *avec le poids* w, noté $t_1 \cdot t_2 \xrightarrow{w}_p t_1 \cdot \alpha \cdot t_2$, ssi $t_1 \cdot t_2 \xrightarrow{w}_R t_1 \cdot \alpha \cdot t_2$ et p est la position concrète de t_2 dans t.

Proposition 100. *Soit une transformation d'abstraction pondérée saine* $R_{\rightsquigarrow}$ *sur le semi-anneau tropical* $\left(S, \oplus, \otimes, \overline{0}, \overline{1}\right)$. *Soit deux termes* $t \in T_{Trace}(\mathcal{F}_\Sigma)$ *et* $t' \in T_{Trace}(\mathcal{F})$ *et deux séquences d'abstractions par* $R_{\rightsquigarrow}$ *:*

$$t \overset{w_1}{\rightsquigarrow}_R \ldots \overset{w_n}{\rightsquigarrow}_R t'$$

et :

$$t \overset{w'_1}{\rightsquigarrow}_R \ldots \overset{w'_{n'}}{\rightsquigarrow}_R t'.$$

Alors $n = n'$ *et :*

$$w_1 \otimes \ldots \otimes w_n = w'_1 \otimes \ldots \otimes w'_{n'}.$$

Démonstration. Par définition de $\rightsquigarrow_R$, dans le semi-anneau tropical, comme w_i est le minimum des w_i^j tels que $t_i \overset{w_i^j}{\rightarrow} t_{i+1}$, il existe deux séquences de réductions par R :

$$t = t_1 \xrightarrow{w_1}_{p_1} \ldots \xrightarrow{w_n}_{p_n} t'$$

et :

$$t \xrightarrow{w'_1}_{p'_1} \ldots \xrightarrow{w'_{n'}}_{p'_{n'}} t'$$

où $p_1, \ldots, p_n$ et $p'_1, \ldots, p'_{n'}$ désignent les positions concrètes auxquelles sont réalisées les réductions.

Les deux séquences transforment t en t' donc la seconde séquence insère les mêmes actions abstraites, aux mêmes positions concrètes, que la première séquence, mais dans un ordre différent. Autrement dit, $n = n'$ et il existe une permutation $\sigma : [1..n] \to [1..n]$ telle que : $\forall i \in [1..n]\,,\, p'_{\sigma(i)} = p_i$.

De plus, comme par définition de $\rightsquigarrow_R$, les poids w_i et w'_i sont, respectivement, les plus petits poids permettant d'effectuer la i-ème abstraction, pour tout i dans $[1..n]$, on a bien : $w_1 \otimes \ldots \otimes w_n = w'_1 \otimes \ldots \otimes w'_{n'}$. □

On en déduit le corollaire suivant, dans le semi-anneau tropical.

Corollaire 101. *Soit une transformation d'abstraction pondérée saine $R_{\rightsquigarrow}$ sur le semi-anneau tropical $\left(S, \oplus, \otimes, \overline{0}, \overline{1}\right)$. Soit deux termes $t \in T_{Trace}\left(\mathcal{F}_\Sigma\right)$ et $t' \in T_{Trace}\left(\mathcal{F}\right)$. Pour toute séquence d'abstractions $t \overset{w_1}{\rightsquigarrow} \ldots \overset{w_n}{\rightsquigarrow} t'$ par $R_{\rightsquigarrow}$, on a :*

$$w_1 \otimes \ldots \otimes w_n = R^*_{\rightsquigarrow}\left(t, t'\right).$$

Démonstration. $R^*_{\rightsquigarrow} = \bigcup_{i \in \mathbb{N}} R^i_{\rightsquigarrow}$ donc $R^*_{\rightsquigarrow}\left(t, t'\right) = \min\limits_{i \in \mathbb{N}} R^i_{\rightsquigarrow}\left(t, t'\right)$.

De plus, $R^i_{\rightsquigarrow}\left(t, t'\right) = \min\limits_{t \overset{w_1}{\rightsquigarrow}_R \ldots \overset{w_i}{\rightsquigarrow}_R t'} w_1 \otimes \ldots \otimes w_i$ donc :

$$R^*_{\rightsquigarrow}\left(t, t'\right) = \min_{i \in \mathbb{N},\, t \overset{w_1}{\rightsquigarrow}_R \ldots \overset{w_i}{\rightsquigarrow}_R t'} w_1 \otimes \ldots \otimes w_i.$$

Le résultat découle alors de la proposition précédente. □

Étendons la définition de la réduction à une position concrète (Définition 99) à l'abstraction à une position concrète.

Définition 102 (Abstraction à une Position Concrète)**.** Soit R un système d'abstraction pondéré sur un semi-anneau $\left(S, \oplus, \otimes, \overline{0}, \overline{1}\right)$. On dit qu'une trace $t = t_1 \cdot t_2$ est *abstraite par R (resp. R_u) en $t_1 \cdot \alpha \cdot t_2$ à la position concrète p avec un poids w*, noté $t_1 \cdot t_2 \overset{w}{\rightsquigarrow}_p t_1 \cdot \alpha \cdot t_2$, ssi $t_1 \cdot t_2 \overset{w}{\rightsquigarrow} t_1 \cdot \alpha \cdot t_2$ par R *(resp. R_u)* et p est la position concrète de t_2 dans t.

Lemme 103. *Soit R un système d'abstraction pondéré sur le semi-anneau tropical $\left(S, \oplus, \otimes, \overline{0}, \overline{1}\right)$. Soit une trace $t \in T_{Trace}\left(\mathcal{F}\right)$ et des actions abstraites $\alpha_1, \ldots, \alpha_k \in T_{Action}\left(\mathcal{F}_\Gamma\right)$. Soit une chaîne d'abstraction pondérée par R* (resp. *R_u) depuis t, de la forme $t \overset{w_1}{\to}{}^* t_1 \cdot t'_1 \overset{w'_1}{\to}_{p_1} t_1 \cdot \alpha_1 \cdot t'_1 \overset{w_2}{\to}{}^* t_2 \cdot t'_2 \overset{w'_2}{\to}_{p_2} t_2 \cdot \alpha_2 \cdot t'_2 \overset{w_3}{\to}{}^* \ldots \overset{w_k}{\to}{}^* t_k \cdot t'_k \overset{w'_k}{\to}_{p_k} t_k \cdot \alpha_k \cdot t'_k$ où on distingue k étapes d'abstraction. Alors, on a la séquence de réductions suivantes par R* (resp. *R_u*) *:*

$$\exists u_1, \ldots u_k, u'_1, \ldots u'_k \in T_{Trace}\left(\mathcal{F}\right),$$
$$t \overset{w'_1}{\to}_{p_1} u_1 \cdot \alpha_1 \cdot u'_1 \overset{w'_2}{\to}_{p_2} u_2 \cdot \alpha_2 \cdot u'_2 \overset{w'_3}{\to}_{p_3} \ldots \overset{w'_k}{\to}_{p_k} u_k \cdot \alpha_k \cdot u'_k.$$

Démonstration. Par induction sur la longueur l de la dérivation $t \to^* t_k \cdot \alpha_k \cdot t'_k$.

- Pour le cas $l = 1$, on a : $t \overset{w_1}{\to}_{p_1} t_1 \cdot \alpha_1 \cdot t'_1$. On définit donc $u_1 = t_1$ et $u'_1 = t'_1$.
- Pour l'étape générale d'induction, supposons que nous avons la propriété pour $l = n$. On montre la propriété pour $l = n + 1$. Par l'hypothèse d'induction appliquée à $t \overset{w_1}{\to}^* t_1 \cdot t'_1 \overset{w'_1}{\to}_{p_1} t_1 \cdot \alpha_1 \cdot t'_1 \overset{w_2}{\to}^* t_2 \cdot t'_2 \ldots \overset{w_k}{\to}^* t_k \cdot t'_k \overset{w'_k}{\to}_{p_k} t_k \cdot \alpha_k \cdot t'_k$, on a : $\exists u_1, \ldots u_k, u'_1, \ldots, u'_k \in T_{Trace}(\mathcal{F})$, $t \overset{w'_1}{\to}_{p_1} u_1 \cdot \alpha_1 \cdot u'_1 \overset{w'_2}{\to}_{p_2} \ldots \overset{w'_k}{\to}_{p_k} u_k \cdot \alpha_k \cdot u'_k$.

  Pour $l = n + 1$, la chaîne de longueur n est étendue par $t_k \cdot \alpha_k \cdot t'_k \overset{w_{k+1}}{\to}^* t_{k+1} \cdot t'_{k+1} \overset{w'_{k+1}}{\to} t_{k+1} \cdot \alpha_{k+1} \cdot t'_{k+1}$.
  On veut réécrire $u_k \cdot \alpha_k \cdot u'_k$ en $u_{k+1} \cdot \alpha_{k+1} \cdot u'_{k+1}$.

  Or l'existence de la réduction $t_{k+1} \cdot t'_{k+1} \overset{w'_{k+1}}{\to} t_{k+1} \cdot \alpha_{k+1} \cdot t'_{k+1}$ entraîne l'existence d'une occurrence du behavior pattern B_{k+1} dans $t_{k+1} \cdot t'_{k+1}$. Cette occurrence apparaît également dans $u_k \cdot \alpha_k \cdot u'_k$ et peut donc être abstraite à la même position concrète p_{k+1} avec le même poids w'_{k+1}, d'où l'existence de termes u_{k+1} et u'_{k+1} tels que : $u_k \cdot \alpha_k \cdot u'_k \overset{w'_{k+1}}{\to}_{p_{k+1}} u_{k+1} \cdot \alpha_{k+1} \cdot u'_{k+1}$.

□

Lemme 104. *Soit R un système d'abstraction pondéré sur le semi-anneau tropical $(S, \oplus, \otimes, \overline{0}, \overline{1})$. Soit une trace $t \in T_{Trace}(\mathcal{F})$ et des actions abstraites $\alpha_1, \ldots, \alpha_k \in T_{Action}(\mathcal{F}_\Gamma)$. Soit une chaîne d'abstraction pondérée depuis t par $R_\rightsquigarrow$, de la forme $t \overset{w_1}{\rightsquigarrow}^* t_1 \cdot t'_1 \overset{w'_1}{\rightsquigarrow}_{p_1} t_1 \cdot \alpha_1 \cdot t'_1 \overset{w_2}{\rightsquigarrow}^* t_2 \cdot t'_2 \overset{w'_2}{\rightsquigarrow}_{p_2} t_2 \cdot \alpha_2 \cdot t'_2 \overset{w_3}{\rightsquigarrow}^* \ldots \overset{w_k}{\rightsquigarrow}^* t_k \cdot t'_k \overset{w'_k}{\rightsquigarrow}_{p_k} t_k \cdot \alpha_k \cdot t'_k$ où on distingue k étapes d'abstraction. Alors :*

$$\exists u_1, \ldots u_k, u'_1, \ldots u'_k \in T_{Trace}(\mathcal{F}),$$
$$t \overset{w'_1}{\rightsquigarrow}_{p_1} u_1 \cdot \alpha_1 \cdot u'_1 \overset{w'_2}{\rightsquigarrow}_{p_2} u_2 \cdot \alpha_2 \cdot u'_2 \overset{w'_3}{\rightsquigarrow}_{p_3} \ldots \overset{w'_k}{\rightsquigarrow}_{p_k} u_k \cdot \alpha_k \cdot u'_k.$$

Démonstration. Par définition de $\rightsquigarrow_R$, sur le semi-anneau tropical (où $\oplus = \min$), il existe une séquence de réductions $t \overset{w_1}{\to}^* t_1 \cdot t'_1 \overset{w'_1}{\to}_{p_1} t_1 \cdot \alpha_1 \cdot t'_1 \overset{w_2}{\to}^* t_2 \cdot t'_2 \overset{w'_2}{\to}_{p_2} t_2 \cdot \alpha_2 \cdot t'_2 \overset{w_3}{\to}^* \ldots \overset{w_k}{\to}^* t_k \cdot t'_k \overset{w'_k}{\to}_{p_k} t_k \cdot \alpha_k \cdot t'_k$.

D'après le Lemme 103, on a la séquence de réductions suivante par R :

$$\exists u_1, \ldots u_k, u'_1, \ldots u'_k \in T_{Trace}(\mathcal{F}),$$
$$t \overset{w'_1}{\to}_{p_1} u_1 \cdot \alpha_1 \cdot u'_1 \overset{w'_2}{\to}_{p_2} u_2 \cdot \alpha_2 \cdot u'_2 \overset{w'_3}{\to}_{p_3} \ldots \overset{w'_k}{\to}_{p_k} u_k \cdot \alpha_k \cdot u'_k.$$

On va montrer, par contradiction, que pour chaque réduction $u_{i-1} \cdot \alpha_{i-1} \cdot u'_{i-1} \overset{w'_i}{\to}_{p_i} u_i \cdot \alpha_i \cdot u'_i$ par R avec $i \in [1..k]$, le poids w'_i est le poids minimum d'une telle réduction et que l'on a donc, par la Définition 86 de la transformation d'abstraction pondérée $\rightsquigarrow_R$ dans le semi-anneau tropical (où $\oplus = \min$) : $u_{i-1} \cdot \alpha_{i-1} \cdot u'_{i-1} \overset{w'_i}{\rightsquigarrow}_{p_i} u_i \cdot \alpha_i \cdot u'_i$.

Supposons qu'il existe un $i \in [1..k]$ tel qu'il existe une réécriture $u_{i-1} \cdot \alpha_{i-1} \cdot u'_{i-1} \overset{w''_i}{\to}_{p_i} u_i \cdot \alpha_i \cdot u'_i$ avec $w''_i < w'_i$. Cela implique qu'il existe une occurrence du behavior pattern B_i dans $u_{i-1} \cdot \alpha_{i-1} \cdot u'_{i-1}$ telle qu'on puisse appliquer une règle de réécriture de poids strictement inférieur à w'_i. Donc cette règle pouvait également être appliquée au terme $t_i \cdot t'_i$, à la même position concrète : $t_i \cdot t'_i \overset{w''_i}{\to}_{p_i} t_i \cdot \alpha_i \cdot t'_i$. Or, par hypothèse, $t_i \cdot t'_i \overset{w'_i}{\rightsquigarrow}_R t_i \cdot \alpha_i \cdot t'_i$ et on est dans le semi-anneau tropical où $\oplus = \min$, donc, par définition de la transformation d'abstraction pondérée $\rightsquigarrow_R$ (cf. Définition 86), w'_i doit être le plus petit poids d'une réécriture de $t_i \cdot t'_i$ en $t_i \cdot \alpha_i \cdot t'_i$, contredisant l'hypothèse $w''_i < w'_i$.

On en déduit que pour tout $i \leq k$, w'_i est le plus petit poids d'une réécriture de $u_{i-1} \cdot \alpha_{i-1} \cdot u'_{i-1}$ en $u_i \cdot \alpha_i \cdot u'_i$. Donc, d'après la Définition 86 de la transformation d'abstraction pondérée $\rightsquigarrow_R$, on a bien, dans le semi-anneau tropical, par R :

$$t \overset{w'_1}{\rightsquigarrow}_{p_1} u_1 \cdot \alpha_1 \cdot u'_1 \overset{w'_2}{\rightsquigarrow}_{p_2} u_2 \cdot \alpha_2 \cdot u'_2 \overset{w'_3}{\rightsquigarrow}_{p_3} \ldots \overset{w'_k}{\rightsquigarrow}_{p_k} u_k \cdot \alpha_k \cdot u'_k.$$

□

On peut maintenant prouver que les comportements malicieux considérés en pratique ont la propriété de (m, n)-complétude.

Théorème 105. *Soit Y un ensemble de variables de sorte Data. Soit $\alpha_1, \ldots, \alpha_m \in T_{Action}(\mathcal{F}_\Gamma, Y)$.*

Alors le comportement abstrait $M := \exists Y. \alpha_1 \odot \alpha_2 \odot \ldots \odot \alpha_m$ a la propriété de $(m, 0)$-complétude.

Démonstration. Soit $\varphi_M = \exists Y. \mathbf{F}(\alpha_1 \odot \alpha_2 \odot \ldots \odot \alpha_m)$.

Il faut montrer :

$$\begin{gathered} L \Cap_{\preceq \rho} M \\ \Leftrightarrow \\ \exists t \in L, \exists t' \in T_{Trace}(\mathcal{F}), \exists i \leq m,\ R^i_{\rightsquigarrow}(t, t') \preceq \rho \\ \text{et} \\ t'|_\Gamma \models \varphi_M. \end{gathered}$$

D'abord, par la sémantique de FOLTL, une trace $t \in T_{Trace}(\mathcal{F}_\Gamma)$ valide la formule $\exists Y.\, \mathbf{F}(\alpha_1 \odot \alpha_2 \odot \ldots \odot \alpha_m)$ ssi t s'écrit $t = t'_1 \cdot t'_2$ avec $t'_1, t'_2 \in T_{Trace}(\mathcal{F}_\Gamma)$ et t'_2 validant la formule $\exists Y.\, \alpha_1 \odot \alpha_2 \odot \ldots \odot \alpha_m$. Puis t'_2 valide la formule $\exists Y.\, \alpha_1 \odot \alpha_2 \odot \ldots \odot \alpha_m$ ssi il existe une instanciation $\sigma_Y \in Inst_Y$ telle que t'_2 valide la formule $\alpha_1\sigma_Y \odot \alpha_2\sigma_Y \odot \ldots \odot \alpha_m\sigma_Y$, équivalente à la formule $\alpha_1\sigma_Y \wedge \mathbf{X}(\top\, \mathbf{U}\, \alpha_2\sigma_Y \wedge \mathbf{X}(\top\, \mathbf{U}\, \ldots \wedge \mathbf{X}(\top\, \mathbf{U}\, \alpha_m\sigma_Y)))$. Donc la trace t'_2 valide les formules $\alpha_1\sigma_Y$ et $\mathbf{X}(\top\, \mathbf{U}\, \alpha_2\sigma_Y \wedge \mathbf{X}(\top\, \mathbf{U}\, \ldots \wedge \mathbf{X}(\top\, \mathbf{U}\, \alpha_m\sigma_Y)))$. La trace t'_2 est donc de la forme :

$$t'_2 = \alpha_1\sigma_Y \cdot t_1 \cdot \alpha_2\sigma_Y \cdot t_2 \cdots \alpha_m\sigma_Y \cdot t_m$$

où $t_1, \ldots, t_m \in T_{Trace}(\mathcal{F}_\Gamma)$.

$\Rightarrow$: Par définition de l'exhibition de M par L, il existe une trace $t \in L$ avec une forme partiellement abstraite $\hat{t}$ par $R_{\rightsquigarrow}$ telle que $\hat{t}\big|_\Gamma \models \varphi_M$, $w = R^*_{\rightsquigarrow}(t, \hat{t}) \preceq \rho$ et $\forall t'' \in R^*_u(\hat{t})\big|_\Gamma$, $t'' \models \varphi_M$.

Étant donné que $\hat{t}\big|_\Gamma \models \varphi_M$, il existe une instanciation $\sigma_Y \in Inst_Y$ telle que :

$$\hat{t} = t_0 \cdot \alpha_1\sigma_Y \cdot t_1 \cdot \alpha_2\sigma_Y \cdot t_2 \cdots \alpha_m\sigma_Y \cdot t_m$$

où $t_0, \ldots, t_m \in T_{Trace}(\mathcal{F})$.

Par le Corollaire 101, toute séquence de transformations $t \overset{w_1}{\rightsquigarrow}_R \ldots \overset{w_n}{\rightsquigarrow}_R \hat{t}$ a le poids $w = w_1 \otimes \ldots \otimes w_n$. Puisque par hypothèse $\hat{t}$ est une forme partiellement abstraite de t, il existe au moins une telle séquence.

Ensuite, en appliquant le Lemme 104 à cette séquence, il existe $u_0, \ldots, u_m \in T_{Trace}(\mathcal{F})$ tels que t est transformé par R en une trace $t' = u_0 \cdot \alpha_1\sigma_Y \cdot u_1 \cdots \alpha_m\sigma_Y \cdot u_m$ en exactement m étapes avec un poids $w' = w_{i_1} \otimes \ldots \otimes w_{i_m}$, pour une certaine séquence d'indices distincts $(i_j)_j$ dans $[1..n]$.

Par conséquent, dans le semi-anneau tropical : $R^m(t, t') \preceq w'$ et $w' \preceq w \preceq \rho$. De plus : $t'|_\Gamma \models \varphi_M$.

$\Leftarrow$: Soit $t \in L$, $i \leq m$ et $t' \in T_{Trace}(\mathcal{F})$ une forme partiellement abstraite d'une trace de L telle que : $R^i(t, t') \preceq \rho$ et $t'|_\Gamma \models \varphi_M$.

Alors t' peut s'écrire $t' = t_0 \cdot \alpha_1\sigma_Y \cdot t_1 \cdots \alpha_m\sigma_Y \cdot t_m$, où $t_0, \ldots, t_m \in T_{Trace}(\mathcal{F})$ et $\sigma_Y \in Inst_Y$. Clairement, toute abstraction future de t' par R_u sera toujours de la forme $u_0 \cdot \alpha_1\sigma_X \cdot u_1 \cdots \alpha_m\sigma_X \cdot u_m$ et validera donc φ_M. Donc t' vérifie la condition de la Définition 92, entraînant : $L \Cap_{\preceq\rho} M$. $\square$

Pour un behavior pattern λ, notons R_λ la restriction de la transformation d'abstraction pondérée R à l'abstraction par rapport à λ. On dit que deux behavior patterns λ et λ' sont *indépendants* ssi : $R_\lambda; R_{\lambda'} = R_{\lambda'}; R_\lambda$.

Exemple 106. Soit $\lambda := a \cdot c$ et $\lambda' := b \cdot c$ deux behavior patterns tels que l'abstraction insère le symbole d'abstraction après l'action c. La trace $a \cdot b \cdot c$ est abstraite en $a \cdot b \cdot c \cdot \lambda' \cdot \lambda$ par $R_\lambda; R_{\lambda'}$ et en $a \cdot b \cdot c \cdot \lambda \cdot \lambda'$ par $R_{\lambda'}; R_\lambda$ donc ces deux behavior patterns ne sont pas indépendants.

On a alors le résultat suivant.

Théorème 107. *Soit* $M := \exists Y.\, \lambda_1(\overline{x_1}) \wedge \neg(\exists Z.\, \lambda_2(\overline{x_2}))\, \mathbf{U}\, \lambda_3(\overline{x_3})$ *un comportement abstrait où* Y *et* Z *sont des ensembles disjoints de variables de sorte Data et où* $\lambda_2 \neq \lambda_1$, $\lambda_2 \neq \lambda_3$ *et* λ_2 *est indépendant de* λ_3.

Alors M *a la propriété de* $(2 + \rho/w_2, 1)$*-complétude.*

Démonstration. Soit $\varphi_M = \exists Y.\, \mathbf{F}(\lambda_1\,(\overline{x_1}) \wedge \neg\,(\exists Z.\, \lambda_2\,(\overline{x_2}))\ \mathbf{U}\, \lambda_3\,(\overline{x_3}))$.

Notons α_1, α_2 et α_3 les actions $\lambda_1\,(\overline{x_1})$, $\lambda_2\,(\overline{x_2})$ et $\lambda_3\,(\overline{x_3})$, respectivement.

Soit $L \subseteq T_{Trace}\,(\mathcal{F}_\Sigma)$ un ensemble de traces. Il faut montrer :

$$\begin{gathered} L \Cap_{\preceq \rho} M \\ \Leftrightarrow \\ \exists t \in L, \exists t' \in T_{Trace}\,(\mathcal{F})\,, \exists i \leq 2 + \rho/w_2,\ R^i_{\rightsquigarrow}(t, t') \preceq \rho \\ \text{et} \\ \forall t'' \in R^{\leq 1}_u\,(t')\,,\ t''|_\Gamma \models \varphi_M. \end{gathered}$$

$\Rightarrow$: Par définition de l'exhibition de M par L, il existe une trace $t \in L$ avec une forme partiellement abstraite $\hat{t}$ par R telle que $\hat{t}\big|_\Gamma$ valide φ_M, $w = R^*_{\rightsquigarrow}(t, \hat{t}) \preceq \rho$ et $\forall t'' \in R^*_u(\hat{t})$, $t''|_\Gamma \models \varphi_M$. Il existe donc une instanciation $\sigma_Y \in Inst_Y$ telle que :

$$\hat{t} = t_1 \cdot \alpha_1 \sigma_Y \cdot t_2 \cdot \alpha_3 \sigma_Y \cdot t_3$$

où $t_1, t_2, t_3 \in T_{Trace}\,(\mathcal{F})$ et t_2 ne contient aucune instance de $\alpha_2 \sigma_Y$.

De plus, on décompose t_2 de façon à identifier toutes les occurrences possibles de $\overline{\alpha_2}\sigma_Y$:

$$t_2 = t_2^1 \cdot \overline{\alpha_2}\sigma_Y \sigma_{Z,1} \cdot t_2^2 \cdots t_2^n \cdot \overline{\alpha_2}\sigma_Y \sigma_{Z,n} \cdot t_2^{n+1}$$

où $t_2^1 \ldots, t_2^{n+1} \in T_{Trace}\,(\mathcal{F})$, $\sigma_{Z,1}, \ldots, \sigma_{Z,n} \in Inst_Z$ et aucune instance de $\overline{\alpha_2}\sigma_Y$ n'apparaît dans $t_2^1 \ldots, t_2^{n+1}$.

Observons que $n \leq \rho/w_2$. En effet, si $w_2 = \overline{0}$, alors aucune instance de $\overline{\alpha_2}$ ne peut apparaître, tandis que si $w_2 \neq \overline{0}$, chaque instance de $\overline{\alpha_2}\sigma_Y$ ajoute un poids d'au moins w_2 au poids final de $\hat{t}$, qui doit être inférieur à ρ.

On définit d'abord un terme t' avec un poids inférieur à ρ dans $R^i_{\rightsquigarrow}\,(t)$ pour un certain $i \leq 2 + \rho/w_2$ tel que t' contienne la même occurrence $\alpha_1 \sigma_Y \cdot$

$\overline{\alpha_2}\sigma_Y\sigma_{Z,1}\cdots\overline{\alpha_2}\sigma_Y\sigma_{Z,n}\cdot\alpha_3\sigma_Y$ de M que $\hat{t}$ et on montre ensuite que ses abstractions futures jusqu'à l'ordre 1 réalisent toujours M.

Par le Corollaire 101, toute séquence de transformations par $\rightsquigarrow_R$ de t à $\hat{t} = t_1\cdot\alpha_1\sigma_Y\cdot t_2^1\cdot\overline{\alpha_2}\sigma_Y\sigma_{Z,1}\cdot t_2^2\cdots t_2^n\cdot\overline{\alpha_2}\sigma_Y\sigma_{Z,n}\cdot t_2^{n+1}\cdot\alpha_3\sigma_Y\cdot t_3$ a le poids $w = R^*(t,\hat{t}) \preceq \rho$. Étant donné que $w \preceq \rho \neq \overline{0}$, il existe au moins une telle séquence.

Ensuite, en appliquant le Lemme 104 à cette séquence, il existe des traces u_1, u_2^1, ..., u_2^{n+1}, u_3 et un poids w' tels que : $t \overset{w'}{\rightsquigarrow}{}_R^* u_1\cdot\alpha_1\sigma_Y\cdot u_2^1\cdot\overline{\alpha_2}\sigma_Y\sigma_{Z,1}\cdot u_2^2\cdots u_2^n\cdot\overline{\alpha_2}\sigma_Y\sigma_{Z,n}\cdot u_2^{n+1}\cdot\alpha_3\sigma_Y\cdot u_3$ en $n+2$ étapes. De plus, dans le semi-anneau tropical : $w' \preceq w \preceq \rho$.

On définit donc :

$$t' = u_1\cdot\alpha_1\sigma_Y\cdot u_2^1\cdot\overline{\alpha_2}\sigma_Y\sigma_{Z,1}\cdot u_2^2\cdots u_2^n\cdot\overline{\alpha_2}\sigma_Y\sigma_{Z,n}\cdot u_2^{n+1}\cdot\alpha_3\sigma_Y\cdot u_3.$$

La trace $t \in L$ étant concrète, $u_1, u_2^1, \ldots, u_2^{n+1}, u_3$ ne contiennent aucune action abstraite, donc $t'|_\Gamma = \alpha_1\sigma_Y\cdot\overline{\alpha_2}\sigma_Y\sigma_{Z,1}\cdots\overline{\alpha_2}\sigma_Y\sigma_{Z,n}\cdot\alpha_3\sigma_Y \models \varphi_M$. Finalement, soit $i = n+2$. Alors, $t \overset{w'}{\rightsquigarrow}{}_R^i t'$ et, par le Corollaire 101, $R^i_{\rightsquigarrow}(t,t')$ est le poids de toute séquence de transformations de t dans t' par R, donc $R^i_{\rightsquigarrow}(t,t') = w'$. De plus, on a observé précédemment que $n \leq \rho/w_2$, donc $i \leq 2+\rho/w_2$. Ainsi, avec notre observation précédente que $w' \preceq \rho$:

$$\exists t' \in T_{Trace}(\mathcal{F}), \exists i \leq 2+\rho/w_2,\ R^i(t,t') \preceq \rho.$$

On montre maintenant que : $\forall t'' \in R_{\overline{u}}^{\leq 1}(t'),\ t''|_\Gamma \models \varphi_M$. On a déjà : $t'|_\Gamma \models \varphi_M$. Supposons qu'il existe un $t'' \in R_u(t')|_\Gamma$ tel que $t'' \not\models \varphi_M$, autrement dit tel que t' puisse être réécrit par R_u de telle manière qu'une action $\alpha_2\sigma_Y\sigma'_Z$ soit insérée dans un sous-terme u_2^i de t' pour une certaine instanciation $\sigma'_Z \in Inst_Z$. Soit p_1 la position concrète de $\alpha_1\sigma_Y$ dans t' et p_3 la position concrète de $\alpha_3\sigma_Y$ dans t'. L'occurrence du behavior pattern λ_2 liée à cette insertion apparaît dans t et peut être abstraite dans t à la même position concrète, c'est-à-dire entre p_1 exclus et p_3 inclus. Donc, il y a deux cas possibles pour $\hat{t}$:

- Cette occurrence a déjà été abstraite dans $\hat{t}$, donc une action $\alpha_2\sigma_Y\sigma'_Z$ ou $\overline{\alpha_2}\sigma_Y\sigma'_Z$ a été insérée à une position concrète entre p_1 exclus et p_3 inclus dans un terme de la dérivation abstraite de t à $\hat{t}$. Étant donné que les behavior patterns λ_2 et λ_3 sont indépendants, leurs abstractions ne peuvent pas avoir lieu à la même position concrète, donc l'occurrence n'a pas pu être abstraite à la position concrète p_3. L'action abstraite a donc été insérée à une position concrète entre p_1 exclus et p_3 exclus. Elle apparaît donc dans t_2.

Finalement, puisque t_2 ne peut, par hypothèse, contenir aucune instance de $\alpha_2\sigma_Y$, l'action abstraite insérée doit être l'une des actions $\overline{\alpha_2}\sigma_Y\sigma_{Z,i}$ identifiées dans $\hat{t}$. Mais lorsque l'on a appliqué le Lemme 104 pour construire t', nous avons déjà considéré l'occurrence associée du behavior pattern λ_2. L'action abstraite insérée ne peut donc pas être une action $\overline{\alpha_2}\sigma_Y\sigma'_Z$ non plus.

– Cette occurrence n'a pas encore été abstraite dans $\hat{t}$. Elle peut alors être abstraite dans $\hat{t}$ à une position concrète entre p_1 exclus et p_3 inclus, autrement dit après une action concrète de t_2 et avant l'action $\alpha_3\sigma_Y$. Il en résulte une trace t'' dont la projection sur Γ ne réalise pas M, ce qui contredit l'hypothèse sur $\hat{t}$: $\forall t'' \in R_u^*(\hat{t}),\ t''|_\Gamma \models \varphi_M$.

$\Leftarrow$: On raisonne par contradiction. Soit $t \in L$, $t' \in T_{Trace}(\mathcal{F})$ et $i \leq 2 + \rho/w_2$ tels que : $R^i(t,t') = w \preceq \rho$ et $\forall t'' \in R_u^{\leq 1}(t')\,,\ t''|_\Gamma \models \varphi_M$. En supposant que L n'exhibe pas M, on construit une trace $t'_1 \in R_u^{\leq 1}(t')$ qui ne réalise pas M, ce qui contredit le fait que : $\forall t'' \in R_u^{\leq 1}(t')\,,\ t''|_\Gamma \models \varphi_M$.

En particulier, $t'|_\Gamma \models \varphi_M$ donc il existe une instanciation $\sigma_Y \in Inst_Y$ telle qu'on puisse décomposer t' en :

$$t' = t_1 \cdot \alpha_1\sigma_Y \cdot t_2 \cdot \alpha_3\sigma_Y \cdot t_3$$

où $t_1, t_2, t_3 \in T_{Trace}(\mathcal{F})$ et t_2 ne contient aucune instance d'$\alpha_2\sigma_Y$.

Supposons que L n'exhibe pas M. Alors, par la Définition 92, étant donné que $R^*(t,t') \preceq R^i(t,t') \preceq \rho$ dans le semi-anneau tropical, il doit exister une trace $t'' \in R_u^*(t')$ telle que $t''|_\Gamma \not\models \varphi_M$. Par définition de M, il doit exister une instanciation $\sigma_Z \in Inst_Z$ telle qu'une action abstraite $\alpha_2\sigma_Y\sigma_Z$ soit insérée dans un terme de la dérivation de t' à t'' par $\rightarrow_{R_u}$, à une position concrète p entre $\alpha_1\sigma_Y$ et $\alpha_3\sigma_Y$. Par le Lemme 104, on aurait pu insérer cette action $\alpha_2\sigma_Y\sigma_Z$ directement dans le terme t', à la même position concrète p, autrement dit entre les actions $\alpha_1\sigma_Y$ et $\alpha_3\sigma_Y$. Soit t''' le terme que nous aurions obtenu. Alors $t'''|_\Gamma \not\models \varphi_M$, ce qui contredit l'hypothèse $\forall t'' \in R_u^{\leq 1}(t')\big|_\Gamma\,,\ t'' \models \varphi_M$. $\square$

Remarquons que l'on peut préciser la borne du théorème précédent en considérant les poids w_1 et w_3 représentant les poids minimaux permettant d'abstraire une occurrence de λ_1 et λ_3 respectivement. Le comportement considéré a alors, de façon assez évidente, la propriété de $(2 + (\rho - w_1 - w_3)/w_2, 1)$-complétude.

On remarque par ailleurs que lorsque les règles d'abstraction du behavior pattern associé à λ_2 sont pondérées par $\overline{1}$ (i.e. l'abstraction est certaine pour λ_2), l'action $\overline{\lambda_2}$ n'est jamais insérée donc $w_2 = \overline{0} = +\infty$ dans le semi-anneau tropical et le comportement a alors la propriété de $(2,1)$-complétude.

6.4 Abstraction Rationnelle

D'après la Définition 87, la transformation d'abstraction est une transformation pondérée de $T_{Trace}(\mathcal{F}) \times T_{Trace}(\mathcal{F}) \to S$. On montre qu'elle est réalisée par un transducteur d'arbres pondéré descendant linéaire non effaçant, i.e. qu'elle est rationnelle.

Pour prouver les résultats de rationalité, on définit un alphabet Γ' distinct de Γ en bijection avec l'alphabet Γ et on adapte l'abstraction pour que l'action abstraite insérée soit définie à partir de Γ'. Cela nous permet d'isoler dans un terme abstrait l'action abstraite insérée des actions abstraites existant déjà avant l'abstraction. Ainsi, notons $\mathcal{F}'$ l'alphabet étendu $\mathcal{F} \cup \Gamma'$. Pour $\lambda \in \Gamma$, on note $\lambda' \in \Gamma'$ son symbole associé, et pour une action $\alpha \in T_{Action}(\mathcal{F}_\Gamma)$, on note $\alpha' \in T_{Action}\left(\mathcal{F}'_{\Gamma'}\right)$ son action associée sur Γ'. On définit finalement les relabelings $l_{\Gamma' \to \Gamma} : T_{Trace}(\mathcal{F}') \to T_{Trace}(\mathcal{F})$ et $l_{\Gamma \to \Gamma'} : T_{Trace}(\mathcal{F}) \to T_{Trace}(\mathcal{F}')$.

La transformation d'abstraction insérant des actions de Γ' est alors définie de la façon suivante.

Définition 108. Soit R une transformation d'abstraction pondérée saine sur un semi-anneau $\left(S, \oplus, \otimes, \overline{0}, \overline{1}\right)$. La *transformation d'abstraction pondérée induite par R sur Γ'* sur le semi-anneau $\left(S, \oplus, \otimes, \overline{0}, \overline{1}\right)$ est la transformation pondérée $R' : T_{Trace}(\mathcal{F}) \times T_{Trace}(\mathcal{F}') \to S$ définie par :

$$\begin{gathered}\forall t_1, t_2 \in T_{Trace}(\mathcal{F}), \forall \alpha \in T_{Action}(\mathcal{F}_\Gamma), \\ t_1 \cdot t_2 \overset{w}{\rightsquigarrow}_{R'} t_1 \cdot \alpha' \cdot t_2 \\ \Leftrightarrow \\ t_1 \cdot t_2 \overset{w}{\rightsquigarrow}_{R} t_1 \cdot \alpha \cdot t_2.\end{gathered}$$

On montre d'abord la rationalité de R' et l'on en déduit ensuite, de façon immédiate, la rationalité de R. Pour cela, on utilise la définition et les lemme suivants.

Définition 109. Soit $\Omega \subseteq \mathcal{F}_a$ et $\Omega' \subseteq \mathcal{F}_a$ deux ensembles disjoints de symboles de fonctions. Soit $s : T_{Trace}(\mathcal{F}_{\Omega'}) \to S$ un ensemble pondéré. La *forme Ω-generalisée* de s, notée $\amalg_\Omega(s)$, est l'ensemble pondéré s' de $T_{Trace}(\mathcal{F}_{\Omega \cup \Omega'})$ défini par :

$$\forall t \in T_{Trace}(\mathcal{F}_{\Omega \cup \Omega'}), \; s'(t) = s\left(t|_{\Omega'}\right).$$

$\amalg_\Omega(s)$ dénote la projection inverse de l'ensemble pondéré s. En un sens, cela revient à injecter aléatoirement des actions de Ω dans les termes de s, sans modifier leur poids.

Par exemple, sur des ensembles non pondérés, soit $\Omega' = \{a, b\}$ et $\Omega = \{c, d\}$ et soit s l'ensemble $s = \{a \cdot b, c\}$. Alors :

$$\begin{aligned} \amalg_{\Omega}(s) = \quad & T_{Trace}(\mathcal{F}_{\Omega}) \cdot a \cdot T_{Trace}(\mathcal{F}_{\Omega}) \cdot b \cdot T_{Trace}(\mathcal{F}_{\Omega}) \\ & \cup \\ & T_{Trace}(\mathcal{F}_{\Omega}) \cdot c \cdot T_{Trace}(\mathcal{F}_{\Omega}). \end{aligned}$$

Lemme 110. *Soit $\Omega \subseteq \mathcal{F}_a$ et $\Omega' \subseteq \mathcal{F}_a$ deux ensembles disjoints de symboles de fonctions et soit $s : T_{Trace}(\mathcal{F}_{\Omega'}) \to S$ un ensemble pondéré reconnu par un automate d'arbres pondéré A. Alors $\amalg_{\Omega}(s)$ est reconnu par un automate d'arbres pondéré de taille $O(|A|)$.*

Démonstration. On définit un transducteur d'arbres pondéré descendant $\tau = (\mathcal{F}_{\Omega'}, \mathcal{F}_{\Omega \cup \Omega'}, \{q_t, q_a, q_d\}, q_t, \Delta)$ tel que : $\|\tau\|(s) = \amalg_{\Omega}(s)$.

Δ est composé des règles suivantes :

- $q_t(\cdot(x_1, x_2)) \to \cdot(q_a(x_1), q_t(x_2))$;
- $q_t(\epsilon) \to \epsilon$;
- Pour tout $f \in \Omega'$ d'arité $k \in \mathbb{N}$, Δ contient la règle :
  $q_a(f(x_1, \ldots, x_k)) \to f(q_d(x_1), \ldots, q_d(x_k))$;
- Pour tout $d \in \mathcal{F}_d$, Δ contient la règle :
  $q_d(d) \to d$;
- Pour tout terme $\alpha \in T_{Action}(\mathcal{F}_{\Omega})$, Δ contient la règle :
  $q_t(x) \to \cdot(\alpha, q_t(x))$.

Le transducteur τ réalise $\amalg_{\Omega}$ et est linéaire et non effaçant et de taille constante par rapport à A. L'application de la Proposition 81 entraîne donc que le langage d'arbres pondéré $\amalg_{\Omega}(s) = \tau(s)$ est reconnu par un automate d'arbres pondéré de taille $O(|\tau| \cdot |A|) = O(|A|)$.

□

Lemme 111. *Soit R une transformation d'abstraction pondérée saine et R' la transformation pondérée induite par R sur Γ'. Soit $I = R'(T_{Trace}(\mathcal{F}))$, l'image de $T_{Trace}(\mathcal{F})$ par $\rightsquigarrow_{R'}$. Alors :*

1. *Pour tous $t \in T_{Trace}(\mathcal{F})$, $t' \in T_{Trace}(\mathcal{F}')$ et pour tout $w \in S \setminus \{\overline{0}\}$:*
$$t \overset{w}{\rightsquigarrow}_{R'} t' \Rightarrow t = t'|_{\Sigma \cup \Gamma}.$$
2. *Pour tout $t' \in T_{Trace}(\mathcal{F}')$:*
$$I(t') = R'(t'|_{\Sigma \cup \Gamma}, t').$$
3. *Pour tout $t' \in T_{Trace}(\mathcal{F}')$:*
$$t'|_{\Sigma \cup \Gamma} \overset{I(t')}{\rightsquigarrow}_{R'} t'.$$

Démonstration. Montrons le premier résultat.

Comme $w \neq \overline{0}$, la transformation R' se fait comme défini dans la Définition 108, donc il existe $t_1, t_2 \in T_{Trace}(\mathcal{F})$ et $\alpha \in T_{Action}(\mathcal{F}')$ tels que : $t = t_1 \cdot t_2$ et $t' = t_1 \cdot \alpha' \cdot t_2$. Donc la trace t' ne diffère de t que par une action abstraite de $T_{Action}\left(\mathcal{F}'_{\Gamma'}\right)$. Or, puisque $t \in T_{Trace}(\mathcal{F})$ et $\alpha' \in T_{Action}(\mathcal{F}_{\Gamma'})$, on a : $t'|_{\Sigma \cup \Gamma} = t$.

Montrons le deuxième résultat.

$$I\left(t'\right) = R'\left(T_{Trace}(\mathcal{F})\right)\left(t'\right) = \bigoplus_{\substack{t \in T_{Trace}(\mathcal{F}), \\ t \overset{w}{\rightsquigarrow}_{R'} t'}} w = \bigoplus_{t \in T_{Trace}(\mathcal{F})} R'\left(t, t'\right).$$

Deux cas sont alors possibles :

- Soit $I(t') = \overline{0}$, auquel cas, par l'égalité ci-dessus : $\forall t \in T_{Trace}(\mathcal{F}),\ R'(t, t') = \overline{0}$ et en particulier : $R'\left(t'|_{\Sigma \cup \Gamma}, t'\right) = \overline{0}$. D'où le résultat.
- Soit $I(t') \neq \overline{0}$, auquel cas, par l'égalité ci-dessus : $\exists t \in T_{Trace}(\mathcal{F}),\ R'(t, t') \neq \overline{0}$, soit $t \overset{R'(t,t')}{\rightsquigarrow}_{R'} t'$ avec $R'(t, t') \neq \overline{0}$. D'après le premier résultat, le seul t tel que $R'(t, t') \neq \overline{0}$ est le terme $t'|_{\Sigma \cup \Gamma}$. Donc la somme $\bigoplus_{t \in T_{Trace}(\mathcal{F})} R'(t, t')$ se réduit au terme $R'\left(t'|_{\Sigma \cup \Gamma}, t'\right)$.

Enfin, le troisième résultat est une reformulation du deuxième résultat. □

Soit un système d'abstraction pondéré R associé à un behavior pattern B. On définit l'ensemble pondéré des instances du membre droit d'une règle $A_i(X) \cdot B_i(X) \cdot y \rightarrow \{ A_i(X) \cdot \lambda(\overline{x}) \cdot B_i(X) \cdot y : w_i,\ A_i(X) \cdot \overline{\lambda}(\overline{x}) \cdot B_i(X) \cdot y : w'_i \}$ comme l'ensemble pondéré associant à chaque terme de $\bigcup_{\sigma \in Inst_{X \cup \{y\}}} A_i(X) \cdot \lambda(\overline{x}) \cdot B_i(X) \cdot y$ un poids w_i et à chaque terme de $\bigcup_{\sigma \in Inst_{X \cup \{y\}}} A_i(X) \cdot \overline{\lambda}(\overline{x}) \cdot B_i(X) \cdot y$ un poids w'_i. On définit alors l'ensemble pondéré des instances des membres droits de règles comme l'union des ensembles pondérés de chaque membre droit de règle. Le poids d'un terme t dans cet ensemble est donc la somme des poids de t dans chaque ensemble d'instances.

Lemme 112. *Soit B un behavior pattern et R une transformation d'abstraction pondérée saine par rapport à B définie par un système d'abstraction dont l'ensemble pondéré des instances des membres droits de règles est reconnu par un automate d'arbres pondéré A_R. Alors :*

- *La transformation pondérée R' induite par R sur Γ' est rationnelle.*
- *Pour tout ensemble pondéré s sur $T_{Trace}(\mathcal{F})$ reconnu par un automate d'arbres pondéré A, $R'(s)$ est reconnu par un automate d'arbres pondéré de taille $O(|A| \cdot |A_R|)$.*

Démonstration. Tout d'abord, pour simplifier la démonstration, on ramène le filtrage des membres gauches de règles lors de l'abstraction à un filtrage couvrant. Remarquons que le filtrage est déjà couvrant à droite. On modifie donc légèrement les ensembles définissant les règles associées au behavior pattern B en concaténant à gauche les ensembles A_i avec $T_{Trace}(\mathcal{F}_\Sigma)$. Par exemple, pour le behavior pattern $\lambda := a$, on définit $A_i = T_{Trace}(\mathcal{F}_\Sigma) \cdot a$ et $B_i = \{\epsilon\}$, au lieu de $A_i = \{a\}$ et $B_i = \{\epsilon\}$. Rappelons qu'un behavior pattern est défini sur $T_{Trace}(\mathcal{F}_\Sigma)$. Cette modification n'altère pas la transformation d'abstraction, elle a un impact constant sur la taille de l'automate A_R et elle nous permet surtout de simplifier la définition de la relation de réduction pondérée générée par R (cf. Définition 85) de la façon suivante :

$$\begin{gathered}\forall t, t' \in T_{Trace}(\mathcal{F}), \forall w \in S,\\ t \xrightarrow{w}_R t'\\ \Leftrightarrow\\ \exists \sigma \in Inst_{X\cup\{y\}}, \exists i \in [1..n], \exists \mu \in \Gamma, (\mu, w) \in \left\{(\lambda, w_i), \left(\overline{\lambda}, w_i'\right)\right\},\\ \exists a \in T_{Trace}(\mathcal{F}) \cdot T_{Action}(\mathcal{F}_\Sigma), \exists b \in T_{Trace}(\mathcal{F}),\\ a|_\Sigma \in A_i(X)\sigma, b|_\Sigma \in B_i(X)\sigma, t = a \cdot b \cdot y\sigma \text{ et } t' = a \cdot \mu(\overline{x})\sigma \cdot b \cdot y\sigma.\end{gathered} \tag{6.1}$$

On en déduit le résultat intermédiaire suivant lorsque la trace t est concrète :

$$\begin{gathered}\forall t \in T_{Trace}(\mathcal{F}_\Sigma), \forall t' \in T_{Trace}(\mathcal{F}), \forall w \in S,\\ t \xrightarrow{w}_R t'\\ \Leftrightarrow\\ t' \text{ est une instance d'un membre droit d'une règle de } R \text{ de poids } w.\end{gathered} \tag{6.2}$$

En effet, d'après (6.1), t' s'écrit : $t' = a \cdot \mu(\overline{x})\sigma \cdot b \cdot y\sigma$ avec $a|_\Sigma \in A_i(X)\sigma$ et $b|_\Sigma \in B_i(X)\sigma$. Or $t = a \cdot b \cdot y\sigma$ est concret donc $a|_\Sigma = a$, $b|_\Sigma = b$, $y\sigma \in T_{Trace}(\mathcal{F}_\Sigma)$ et : $t' \in \sigma(A_i(X) \cdot \mu(\overline{x}) \cdot B_i(X) \cdot y)$.

Soit C l'ensemble pondéré des instances de membres droits des règles de R.

On construit un transducteur réalisant $\rightsquigarrow_{R'}$, de la façon suivante.

Soit I l'image de $T_{Trace}(\mathcal{F})$ par $\rightsquigarrow_{R'}$: $I = R'(T_{Trace}(\mathcal{F}))$.

Dans un premier temps, on montre que I est un ensemble régulier qui peut s'exprimer en fonction de l'ensemble pondéré $l_{\Gamma\to\Gamma'}(C)$.

Dans un deuxième temps, on simule la transformation pondérée $\rightsquigarrow_{R'}$ par la composition fonctionnelle de deux transformations :

- une première transformation non pondérée R_{pick}, qui injecte une action α' de $T_{Action}\left(\mathcal{F}'_{\Gamma'}\right)$ dans la trace $t \in T_{Trace}(\mathcal{F})$ à abstraire, à un endroit aléatoire, produisant une trace t' ;
- une seconde transformation pondérée Id_I qui réalise l'identité sur l'ensemble pondéré I construit dans le premier point et qui garantit donc que t' est dans l'image de $T_{Trace}(\mathcal{F})$ par R'.

Traitons le premier point. On exprime l'ensemble $I = R'(T_{Trace}(\mathcal{F}))$ en fonction de l'ensemble C des instances de membres droits de R.

Pour cela, on définit l'ensemble non pondéré $Valid$ des traces vérifiant la condition de santé de la Définition 89 appliquée à R' :

$$\begin{aligned} Valid = \quad & \bigcup_{\alpha \in T_{Action}(\mathcal{F}_\Gamma)} T_{Trace}(\mathcal{F}) \cdot T_{Action}(\mathcal{F}_\Sigma) \cdot \alpha' \cdot \\ & (T_{Trace}(\mathcal{F}) \setminus (T_{Trace}(\mathcal{F}_\Gamma) \cdot \{\alpha, \overline{\alpha}\} \cdot T_{Trace}(\mathcal{F}))) . \end{aligned}$$

Soit deux termes $u, v \in T_{Trace}(\mathcal{F})$ et une action $\alpha \in T_{Action}(\mathcal{F}_\Gamma)$. On note α' son action associée sur $T_{Action}\left(\mathcal{F}'_{\Gamma'}\right)$.

Supposons que l'on a, pour un certain poids non nul $w \in S \setminus \left\{\overline{0}\right\}$:

$$u \cdot v \overset{w}{\rightsquigarrow}_{R'} u \cdot \alpha' \cdot v.$$

Alors, w étant non nul et par définition de R', on a, de façon équivalente :

$$u \cdot v \overset{w}{\rightsquigarrow}_{R} u \cdot \alpha \cdot v.$$

R étant une transformation d'abstraction saine, on sait alors, de façon équivalente, que :

- D'une part, l'action α insérée par R dans $u \cdot v$ vérifie la condition de santé, c'est-à-dire que les actions abstraites qui la suivent ne contiennent pas α ou $\overline{\alpha}$. Autrement dit, par définition de $Valid$:

$$u \cdot \alpha' \cdot v \in Valid.$$

- D'autre part, il existe n étapes de réduction $u \cdot v \overset{w_i}{\rightarrow}_R u \cdot \alpha \cdot v$ et w est donné par : $w = \bigoplus_{i \in [1..n]} w_i$.

On a donc, finalement :

$$\begin{gathered} u \cdot v \overset{w}{\rightsquigarrow}_{R'} u \cdot \alpha' \cdot v \\ \Leftrightarrow \\ u \cdot \alpha' \cdot v \in Valid \text{ et} \\ \text{il existe } n \text{ étapes de réduction } u \cdot v \overset{w_i}{\rightarrow}_R u \cdot \alpha \cdot v \text{ et } w = \bigoplus_{i \in [1..n]} w_i. \end{gathered} \tag{6.3}$$

Considérons une de ces étapes de réduction $u \cdot v \xrightarrow{w_i}_R u \cdot \alpha \cdot v$. Alors, par définition de la relation de réduction R, on a :

$$u \cdot \alpha' \cdot v \in Valid \text{ et } \; u|_\Sigma \cdot v|_\Sigma \xrightarrow{w_i}_R u|_\Sigma \cdot \alpha \cdot v|_\Sigma \,.$$

Réciproquement, lorsque $u|_\Sigma \cdot v|_\Sigma \xrightarrow{w_i}_R u|_\Sigma \cdot \alpha \cdot v|_\Sigma$ alors $u \cdot v \xrightarrow{w_i}_R u \cdot \alpha \cdot v$, à condition que l'action α insérée vérifie la condition de santé, ce qui est vrai car, comme dit précédemment, cela équivaut à requérir $u \cdot \alpha' \cdot v \in Valid$, qui est dans nos hypothèses.

Et donc, en appliquant la formule (6.2), $u|_\Sigma \cdot \alpha \cdot v|_\Sigma$ est de façon équivalente une instance d'un membre droit d'une règle de R de poids w_i.

Pour résumer :

$$\begin{gathered} u \cdot \alpha' \cdot v \in Valid \text{ et } u \cdot v \xrightarrow{w_i}_R u \cdot \alpha \cdot v \\ \Leftrightarrow \\ u|_\Sigma \cdot \alpha \cdot v|_\Sigma \text{ est une instance d'un membre droit} \\ \text{d'une règle de } R \text{ de poids } w_i. \end{gathered}$$

Autrement dit, lorsque $u \cdot \alpha' \cdot v \in Valid$, chaque instance $u|_\Sigma \cdot \alpha \cdot v|_\Sigma$ d'un membre droit d'une règle de R de poids w_i identifie une étape de réduction $u \cdot v \xrightarrow{w_i}_R u \cdot \alpha \cdot v$, et réciproquement.

Or, par définition de C, $C(u|_\Sigma \cdot \alpha \cdot v|_\Sigma)$ est la somme des poids w_i tels que $u|_\Sigma \cdot \alpha \cdot v|_\Sigma$ soit une instance d'un membre droit d'une règle de R de poids w_i. Donc $C(u|_\Sigma \cdot \alpha \cdot v|_\Sigma)$ est aussi la somme des poids w_i de chaque étape de réduction $u \cdot v \xrightarrow{w_i}_R u \cdot \alpha \cdot v$.

En repartant de (6.3), on en déduit :

$$\begin{gathered} u \cdot v \overset{w}{\rightsquigarrow}_{R'} u \cdot \alpha' \cdot v \\ \Leftrightarrow \\ u \cdot \alpha' \cdot v \in Valid \text{ et} \\ w = C(u|_\Sigma \cdot \alpha \cdot v|_\Sigma). \end{gathered}$$

Enfin, on observe que :

$$C(u|_\Sigma \cdot \alpha \cdot v|_\Sigma) = l_{\Gamma \to \Gamma'}(C)(u|_\Sigma \cdot \alpha' \cdot v|_\Sigma). \tag{6.4}$$

En effet, par définition de l'application d'une transformation pondérée à un ensemble pondéré (cf. Section 6.1) :

$$l_{\Gamma \to \Gamma'}(C)(u|_\Sigma \cdot \alpha' \cdot v|_\Sigma) = \bigoplus_{t \in T_{Trace}(\mathcal{F}),\, t \overset{w}{\rightsquigarrow}_{l_{\Gamma \to \Gamma'}} u|_\Sigma \cdot \alpha' \cdot v|_\Sigma} w \otimes C(t)$$

et le seul t pour lequel $t \overset{w}{\rightsquigarrow}_{l_{\Gamma\to\Gamma'}} u|_\Sigma \cdot \alpha' \cdot v|_\Sigma$ avec $w \neq \overline{0}$ est $t = u|_\Sigma \cdot \alpha \cdot v|_\Sigma$ et, dans ce cas, $w = \overline{1}$ car la transformation est un relabeling qui n'altère pas les poids.

Donc finalement, en appliquant (6.4) :

$$\begin{array}{c} u \cdot v \overset{w}{\rightsquigarrow}_{R'} u \cdot \alpha' \cdot v \\ \Leftrightarrow \\ u \cdot \alpha' \cdot v \in Valid \text{ et} \\ w = l_{\Gamma\to\Gamma'}(C)(u|_\Sigma \cdot \alpha' \cdot v|_\Sigma). \end{array} \tag{6.5}$$

Soit Id_{Valid} l'identité sur l'ensemble $Valid$. Montrons que l'ensemble pondéré $I = R'\left(T_{Trace}\left(\mathcal{F}\right)\right)$ est identique à l'ensemble pondéré I' défini par :

$$I' = Id_{Valid}\left(\Pi_\Gamma\left(l_{\Gamma\to\Gamma'}\left(C\right)\right)\right).$$

Tout d'abord, pour tout $t' \in T_{Trace}\left(\mathcal{F}'\right)$ ne s'écrivant pas $t' = u \cdot \alpha' \cdot v$ avec $u, v \in T_{Trace}\left(\mathcal{F}\right)$ et $\alpha' \in T_{Action}\left(\mathcal{F}'_{\Gamma'}\right)$, on observe que :

$$I(t') = I'(t') = \overline{0} \tag{6.6}$$

car d'une part, t' ne peut pas être dans l'image par R' de $T_{Trace}\left(\mathcal{F}\right)$ donc $I(t') = \overline{0}$ et d'autre part, t' ne peut pas être dans $Valid$ donc $I'(t') = \overline{0}$.

Prenons maintenant un terme $t' \in T_{Trace}\left(\mathcal{F}'\right)$ s'écrivant : $t' = u \cdot \alpha' \cdot v$.

Soit un ensemble pondéré s sur $T_{Trace}\left(\mathcal{F}'\right)$. Observons tout d'abord que, par définition de la transformation identité et sachant que $Valid$ est un ensemble non pondéré (i.e. ses élément ont un poids $\overline{1}$), on a :

$$\begin{array}{rl} Id_{Valid}(s)(t') = & \bigoplus_{t \in T_{Trace}(\mathcal{F}\cup\Gamma'),\, t \overset{w}{\rightsquigarrow}_{Id_{Valid}} t'} s(t) \otimes w \\ = & \begin{cases} s(t') & \text{si } t' \in Valid \\ \overline{0} & \text{sinon.} \end{cases} \end{array}$$

En appliquant ce résultat à $I'(t')$, on a :

$$I'(t') = \begin{cases} \amalg_\Gamma (l_{\Gamma\to\Gamma'}(C))(t') & \text{si } t' \in Valid \\ \overline{0} & \text{sinon} \end{cases}$$

$$\Leftrightarrow$$

$$I'(t') = \begin{cases} w & \text{si } t' \in Valid, \text{ si } w = \amalg_\Gamma (l_{\Gamma\to\Gamma'}(C))(u\cdot\alpha'\cdot v) \text{ et si } w \neq \overline{0} \\ \overline{0} & \text{sinon} \end{cases}$$

$$\Leftrightarrow$$

par définition de $\amalg_\Gamma$

$$I'(t') = \begin{cases} w & \text{si } t' \in Valid, \text{ si } w = l_{\Gamma\to\Gamma'}(C)(u|_\Sigma\cdot\alpha'\cdot v|_\Sigma) \text{ et si } w \neq \overline{0} \\ \overline{0} & \text{sinon} \end{cases}$$

$$\Leftrightarrow$$

par (6.5)

$$I'(t') = \begin{cases} w & \text{si } u\cdot v \overset{w}{\rightsquigarrow}_{R'} u\cdot\alpha'\cdot v \text{ et } w \neq \overline{0} \\ \overline{0} & \text{sinon} \end{cases}$$

$$\Leftrightarrow$$

par le Lemme 111

$$I'(t') = \begin{cases} I(u\cdot\alpha'\cdot v) & \text{si } I(u\cdot\alpha'\cdot v) \neq \overline{0} \\ \overline{0} & \text{sinon} \end{cases}$$

Ainsi, avec (6.6), on a bien, sur $T_{Trace}(\mathcal{F}')$: $I' = I$. Donc, par définition de I' :

$$I = Id_{Valid}(\amalg_\Gamma (l_{\Gamma\to\Gamma'}(C))). \tag{6.7}$$

Traitons maintenant le second point. Nous montrons que :

$$R' = R_{pick}; Id_I. \tag{6.8}$$

Rappelons que la transformation R_{pick} est la transformation qui injecte aléatoirement une action de $T_{Action}(\mathcal{F}'_{\Gamma'})$ dans la trace à abstraire. Autrement dit, pour tous $t \in T_{Trace}(\mathcal{F})$ et $t' \in T_{Trace}(\mathcal{F}')$:

$$t \overset{w}{\rightsquigarrow}_{R_{pick}} t'$$

$$\Leftrightarrow$$

$$w = \begin{cases} \overline{1} & \text{si} \quad \exists\alpha' \in T_{Action}(\mathcal{F}'_{\Gamma'}), \exists t_1, t_2 \in T_{Trace}(\mathcal{F}), \\ & \quad\quad t = t_1\cdot t_2, t' = t_1\cdot\alpha'\cdot t_2 \\ \overline{0} & \text{sinon.} \end{cases} \tag{6.9}$$

Par définition de la composition de transformations pondérées, on a donc, pour tous $t \in T_{Trace}(\mathcal{F})$ et $t' \in T_{Trace}(\mathcal{F}')$:

$$t \overset{w}{\rightsquigarrow}_{R_{pick};Id_I} t'$$
$$\Leftrightarrow$$
$$w = \bigoplus_{t'' \in T_{Trace}(\mathcal{F}')} R_{pick}(t, t'') \otimes Id_I(t'', t')$$

Par définition de la transformation identité sur un ensemble, on a, pour tout $t'' \neq t'$: $Id_I(t'', t') = \overline{0}$. On en déduit :

$$t \overset{w}{\rightsquigarrow}_{R_{pick};Id_I} t'$$
$$\Leftrightarrow$$
$$w = R_{pick}(t, t') \otimes Id_I(t', t') = R_{pick}(t, t') \otimes I(t')$$
$$\Leftrightarrow$$
$$\text{par (6.9)}$$
$$w = \begin{cases} \overline{0} & \text{si } R_{pick}(t, t') = \overline{0} \\ I(t') & \text{sinon.} \end{cases}$$
$$\Leftrightarrow$$
$$\text{par le Lemme 111}$$
$$w = \begin{cases} \overline{0} & \text{si } R_{pick}(t, t') = \overline{0} \\ R'(t'|_{\Sigma \cup \Gamma}, t') & \text{sinon.} \end{cases}$$

Or, si $R_{pick}(t, t') = \overline{0}$, alors, par la définition de R_{pick}, t' ne peut pas s'obtenir depuis t en insérant une action abstraire de $T_{Action}(\mathcal{F}'_{\Gamma'})$ dans t donc nécessairement : $R'(t, t') = \overline{0}$. Et si $R_{pick}(t, t') \neq \overline{0}$, alors, par la définition de R_{pick} : $t'|_{\Sigma \cup \Gamma} = t$.

On obtient donc :

$$t \overset{w}{\rightsquigarrow}_{R_{pick};Id_I} t'$$
$$\Leftrightarrow$$
$$w = R'(t, t')$$
$$\Leftrightarrow$$
$$t \overset{w}{\rightsquigarrow}_{R'} t'.$$

Pour récapituler, on a donc, avec (6.7) :

$$I = Id_{Valid}(\amalg_{\Gamma}(l_{\Gamma \to \Gamma'}(C)))$$

et, avec (6.8) :

$$R' = R_{pick}; Id_I.$$

On peut désormais **conclure**.

Tout d'abord, on montre que I est régulier et reconnu par un automate de traces pondéré de taille $O(|A_R|)$. L'homomorphisme $l_{\Gamma\to\Gamma'}$ étant un relabeling, par la Proposition 79, il est réalisé par un transducteur de traces de taille constante. L'ensemble pondéré C étant reconnu par hypothèse par l'automate A_R, par la Proposition 81, l'ensemble pondéré $l_{\Gamma\to\Gamma'}(C)$ est donc reconnu par un automate de traces pondéré de taille $O(|A_R|)$. Par le Lemme 110, l'ensemble pondéré $\amalg_\Gamma(l_{\Gamma\to\Gamma'}(C))$ est donc reconnu par un automate de traces pondéré de taille $O(|A_R|)$. L'ensemble $Valid$ étant reconnu par un automate de taille constante, par la Proposition 80, la transformation Id_{Valid} est réalisée par un transducteur de taille constante. L'ensemble pondéré $I = Id_{Valid}(\amalg_\Gamma(l_{\Gamma\to\Gamma'}(C)))$ est donc reconnu par un automate de traces pondéré de taille $O(|A_R|)$, par la Proposition 81.

Par la Proposition 80, il existe donc un transducteur de traces pondéré réalisant Id_I et de taille $O(|A_R|)$. Enfin, R_{pick} est rationnelle et réalisée par un transducteur de traces pondéré de taille constante.

On en déduit que d'une part, les transformations pondérées rationnelles étant fermées par composition fonctionnelle, la relation $R' = R_{pick}; Id_I$ est rationnelle.

Et d'autre part, pour tout automate de traces pondéré A, l'ensemble pondéré $R'(\|A\|) = Id_I(R_{pick}(\|A\|))$ est reconnu par un automate de traces pondéré de taille $O(|A_R| \cdot |A|)$, en appliquant deux fois la Proposition 81. □

On déduit du Lemme 112 la rationalité de R, que nous exprimons dans le théorème suivant.

Théorème 113. *Soit B un behavior pattern et R une transformation d'abstraction pondérée saine par rapport à B définie par un système d'abstraction dont l'union des ensembles pondérés des instances de membres droits de règles est reconnue par un automate d'arbres pondéré A_R. Alors :*

- *R_u et R_u^{-1} sont rationnelles et réalisées par des transducteurs d'arbres de taille $O(|A_R|)$.*
- *R est rationnelle et, pour tout automate d'arbres pondéré A, $R(\|A\|)$ est reconnu par un automate d'arbres pondéré de taille $O(|A| \cdot |A_R|)$.*

Démonstration. La relation R_u étant une relation d'abstraction non pondérée au sens du chapitre précédent, la rationalité de R_u et de R_u^{-1} et l'existence de deux transducteurs non pondérés de taille $O(|A_{R_u}|) = O(|A_R|)$ les réalisant est une conséquence directe du Théorème 71.

Soit R' la transformation pondérée induite par R sur Γ'. Alors, $R = R'; l_{\Gamma'\to\Gamma}$.

Par la Proposition 79, le relabeling $l_{\Gamma' \to \Gamma}$ est réalisé par un transducteur d'arbres de taille constante.

Par le Lemme 112, R' est rationnelle et, pour tout automate d'arbres pondéré A, l'ensemble pondéré $R'(\|A\|)$ est reconnu par un automate d'arbres pondéré de taille $O(|A| \cdot |A_R|)$. On en déduit que l'ensemble pondéré $R(\|A\|) = l_{\Gamma' \to \Gamma}(R'(\|A\|))$ est reconnu par un automate d'arbres pondéré de taille $O(|A| \cdot |A_R|)$, par la Proposition 81.

Les transformations $l_{\Gamma' \to \Gamma}$ et R' étant rationnelles, la transformation R est de plus rationnelle. □

En utilisant l'ensemble des traces n-exhibant M, on obtient la complexité suivante pour la détection de M, qui reste linéaire en la taille de l'automate reconnaissant l'ensemble de traces du programmes, comme c'était le cas dans l'approche non pondérée.

Théorème 114. *Soit R une transformation d'abstraction pondérée saine sur un semi-anneau $(S, \oplus, \otimes, \overline{0}, \overline{1})$, définie par un système d'abstraction dont l'union des ensembles pondérés des instances de membres droits de règles est reconnue par un automate d'arbres pondéré A_R. Soit M un comportement abstrait régulier ayant la propriété de (m, n)-complétude et A_M un automate d'arbres reconnaissant l'ensemble des traces n-exhibant M.*

Il existe une procédure de détection décidant si un ensemble régulier de traces L, reconnu par un automate d'arbres A, exhibe M par rapport à un seuil $\rho \in S \setminus \{\overline{0}\}$, en temps et espace $O\left(|A_R|^{m \cdot (m+1)/2} \times |A| \times |A_M|\right)$.

Démonstration. La preuve du Théorème 97 reposait sur le résultat suivant :

$$L \Cap_{\preceq \rho} M$$
$$\Leftrightarrow$$
$$\exists t \in T_{Trace}(\mathcal{F}),\ Id_{\|A_M\|}\left(R_{\rightsquigarrow}^{\leq m}(L)\right)(t) \preceq \rho.$$

Pour $i \in [1..m]$, $R_{\rightsquigarrow}^{i}(L)$ est reconnu par un automate d'arbres pondéré de taille $O\left(|A_R|^{i} \times |A|\right)$, en appliquant i fois le Théorème 113. Il existe donc un automate d'arbres pondéré de taille $O\left(|A_R|^{m \cdot (m+1)/2} \times |A|\right)$ reconnaissant $R_{\rightsquigarrow}^{\leq m}(L)$.

D'après la Proposition 80, la transformation $Id_{\|A_M\|}$ est réalisée par un transducteur d'arbres pondéré de taille $O(|A_M|)$.

D'après la Proposition 81, l'ensemble $Id_{\|A_M\|}\left(R_{\rightsquigarrow}^{\leq m}(L)\right)$ est donc reconnu par un automate d'arbres pondéré de taille $O\left(|A_R|^{m \cdot (m+1)/2} \times |A| \times |A_M|\right)$.

Enfin, dans le semi-anneau tropical, la recherche du chemin de plus petit poids dans un automate d'arbres pondéré est linéaire [62]. □

6.5 Conclusion

Le formalisme d'abstraction pondéré présenté a l'avantage de fournir un algorithme de détection de même complexité que dans le cas non pondéré, c'est-à-dire linéaire en la taille de l'automate de traces. Ainsi, sans surcoût, nous pouvons prendre en compte les incertitudes liées à l'abstraction, qu'elles proviennent d'erreurs d'analyse statique ou qu'elles soient localisées dans la réalisation d'une fonctionnalité.

En outre, les définitions des behavior patterns peuvent être affinées en adaptant le poids de l'abstraction en fonction du contexte : si l'occurrence reconnue apparaît dans un contexte propice à la mise en oeuvre de la fonctionnalité associée, on réduira l'incertitude, tandis qu'on l'augmentera dans le cas inverse. Cela revient à adapter les ensembles A_i (contexte passé) et B_i (contexte futur). De même, si le programme est obfusqué, on peut être plus tolérant vis-à-vis de l'incertitude de l'abstraction car le flux de données comportera alors sans doute un plus grand nombre d'erreurs.

Enfin, nous pourrions exprimer le problème de la détection d'un comportement abstrait d'une manière légèrement différente : plutôt que de déterminer si le programme exhibe un comportement avec une probabilité ne dépassant pas un seuil donné a priori, on pourrait calculer la probabilité exacte que le programme exhibe ce comportement. Une telle approche du problème permettrait d'envisager deux scénarios. D'une part, un analyste humain aurait une information plus précise quant au comportement exhibé. D'autre part, supposons que l'exhibition du comportement soit un critère de décision parmi d'autres pour évaluer le caractère malicieux du programme : le poids de ce critère dans la décision finale pourrait être fonction de la probabilité que le programme exhibe ce comportement.

Chapitre 7

Conclusion

L'analyse comportementale traditionnelle opère en général au niveau de l'implantation du comportement malveillant. Pourtant, elle s'intéresse surtout à l'identification d'un comportement donné, indépendamment de sa mise en œuvre technique, autrement dit l'analyse comportementale se situe plus naturellement à un niveau fonctionnel. De plus, comme elle se situe au niveau de la mise en œuvre technique, elle risque d'être plus facilement compromise lors de modifications dans l'implantation des différents composants d'un comportement, que si elle se situait au niveau de la fonction de ces composants.

Ainsi, nous avons défini une forme d'analyse comportementale de programmes qui opère non pas sur les interactions élémentaires d'un programme avec le système mais sur la fonction que le programme réalise.

En outre, l'approche traditionnelle de l'analyse comportementale repose soit sur des techniques d'analyse dynamique, contenant éventuellement une phase d'abstraction des traces capturées, soit sur des techniques d'analyse statique, sans abstraction. Le cas de l'analyse statique avec abstraction n'est pas abordé dans la littérature, en partie parce qu'il entraîne des problèmes d'indécidabilité.

Nous avons donc :

- défini de façon simple, intuitive et formelle des fonctionnalités de base à abstraire et des comportements à détecter en fonction de ces fonctionnalités ;
- proposé un mécanisme d'abstraction applicable à un cadre d'analyse statique ou dynamique, avec des algorithmes pratiques à complexité raisonnable ;
- décrit une technique d'analyse comportementale intégrant ce mécanisme d'abstraction.

Notre méthode est particulièrement adaptée à l'analyse des programmes dans des langages de haut niveau ou dont le code source est connu, pour lesquels l'analyse statique est facilitée : les programmes conçus pour des machines virtuelles comme Java ou .NET, les scripts Web, les extensions de navigateurs, les composants off-the-shelf (COTS).

Le formalisme d'analyse comportementale par abstraction que nous avons proposé repose sur la théorie de la réécriture de mots et de termes, les langages réguliers de mots et de termes et le model checking. Il a les caractéristiques suivantes :

- il permet d'identifier efficacement des fonctionnalités dans des traces et ainsi d'obtenir une représentation des traces à un niveau fonctionnel ;
- il définit les fonctionnalités et les comportements de façon naturelle, à l'aide de formules de logique temporelle, ce qui garantit leur simplicité et leur flexibilité et permet l'utilisation de techniques de model checking pour la détection de ces comportements ;
- il opère sur un ensemble quelconque de traces d'exécution ;
- il prend en compte le flux de données dans les traces d'exécution ;
- il permet, sans perte d'efficacité, de tenir compte de l'incertitude dans l'identification des fonctionnalités.

Pour le construire, nous nous sommes d'abord intéressés à l'abstraction d'ensembles de traces définis comme des langages de mots, en définissant l'abstraction comme la relation induite par un système de réécriture de mots qui remplace, dans une trace, une occurrence d'une fonctionnalité par un symbole abstrait identifiant de façon unique cette fonctionnalité. L'ensemble de traces complètement abstraites, alors calculé par réécriture, est projeté sur l'ensemble Γ des symboles abstraits, produisant un ensemble de traces Γ-abstrait. La détection consiste alors à comparer cet ensemble de traces à un comportement abstrait de référence, ou signature, spécifié comme un langage sur Γ.

En considérant des ensembles de traces réguliers, nous avons proposé un algorithme efficace construisant un automate reconnaissant l'ensemble de traces Γ-abstrait avec un nombre d'états linéaire par rapport au nombre d'états de l'automate de traces initial. Nous avons également proposé un algorithme de détection, en temps linéaire par rapport à la taille de l'automate de traces Γ-abstrait et la taille de l'automate représentant la signature. Nous avons, pour terminer, étayé nos résultats par des expériences sur des automates de traces construits à partir de traces capturées par instrumentation binaire dynamique.

Ensuite, nous avons augmenté la puissance de notre formalisme en représentant les traces par des termes, afin de pouvoir prendre en compte les arguments des appels de librairie lors de l'abstraction et lors de la détection. Nous

avons considéré l'abstraction et la détection sous l'angle du model checking, en constatant que d'une part les fonctionnalités et les comportements de haut niveau sont définis naturellement par des formules de logique temporelle et que d'autre part, l'abstraction puis la détection peuvent être interprétées comme la validation d'une formule de logique temporelle sur des traces. Nous avons alors défini l'abstraction comme une relation induite par un système de réécriture de termes marquant une occurrence de fonctionnalité par insertion d'une action abstraite à son niveau.

En faisant le choix du marquage d'une fonctionnalité plutôt que son remplacement, nous pouvons tenir compte des occurrences de fonctionnalité entrelacées mais, en contrepartie, le calcul de l'ensemble des traces complètement abstraites d'un langage régulier de traces devient indécidable. Aussi, nous avons identifié la (m, n)-complétude, propriété applicable à des familles de comportements abstraits rencontrés en pratique, et qui nous évite d'avoir à calculer cet ensemble. La détection se limite ainsi, sans perte de puissance pour la détection, au calcul d'un ensemble de traces partiellement abstraites puis à la validation de la formule de logique temporelle définissant la signature. Là-encore, pour des comportements ayant cette propriété, nous avons proposé un algorithme efficace de détection, en temps linéaire par rapport à la taille de l'automate de traces. Nous avons montré que ce formalisme s'appliquait au problème important de la fuite d'informations et nous avons étayé nos résultats par des simulations avec l'outil de modélisation CADP.

Enfin, nous avons étendu notre formalisme pour exprimer l'incertitude sur les abstractions, pour pouvoir travailler entre autres lorsque le flux de données calculé par analyse statique est erroné ou lorsqu'une séquence d'actions réalisant une fonctionnalité est détectée sans que tous ses composants aient été reconnus. Nous avons proposé de représenter l'incertitude par un poids défini sur un semi-anneau et nous avons défini l'abstraction comme une transformation pondérée induite par un système de réécriture pondérée de termes. Souhaitant décider si un ensemble de traces exhibe un certain comportement abstrait avec une certitude supérieure à un seuil fixé, nous avons montré que la (m, n)-complétude pouvait être généralisée au cas pondéré. La détection des comportements ayant cette propriété étant décidable, nous avons proposé un algorithme de détection linéaire en la taille de l'automate de traces, sans aucun sur-coût par rapport au cas non pondéré.

7.1 Applications

L'approche d'analyse comportementale que nous avons présentée trouve de multiples applications, au niveau de la détection de comportements comme de l'analyse de programmes.

Dans le contexte de la détection de comportements prédéfinis, on a vu qu'un comportement de haut niveau pouvait être détecté efficacement et qu'un seuil pouvait être défini pour prendre en compte un certain degré de certitude dans le calcul des formes abstraites de traces du programme. Toujours dans le contexte de la détection de programmes malveillants, notre méthode peut aussi apporter des critères de décision pour compléter l'algorithme de détection d'un antivirus : chaque comportement suspect exhibé par le programme peut venir renforcer le résultat d'analyse de l'antivirus. Inversement, la non-exhibition de comportements peut venir renforcer la décision de laisser s'exécuter le programme.

Dans le contexte de l'analyse manuelle ensuite, c'est-à-dire lorsqu'un analyste humain chercher à déterminer et comprendre le comportement du programme, notre méthode peut être appliquée dans les scénarios suivants :

Détection d'un comportement quelconque. Un analyste peut souhaiter déterminer si le programme exhibe un comportement quelconque, potentiellement anodin comme une création de service Windows, un accès à un fichier, etc. On se ramène dans ce cas au cas de la détection de comportements de haut niveau, mais plus par rapport à une base de données de comportement suspects.

Découverte automatique de comportements inconnus. Cette application, outrepassant les objectifs que nous nous étions fixés au départ, nous paraît être particulièrement intéressante. Il s'agit de déterminer automatiquement quels comportements exhibe un programme et pas nécessairement des comportements connus définis dans une base de données. Par exemple, en appliquant deux fois la relation d'abstraction, on obtient toutes les séquences de deux fonctionnalités qui apparaissent dans des traces du programme. Imaginons qu'une de ces séquences soit $\lambda_1(d) \cdot \lambda_2(d)$ pour un certain $d \in \mathcal{F}_d$; cette séquence en tant que telle n'a pas d'intérêt si la donnée d est invalidée entre temps. Supposons alors que l'invalidation de la donnée d corresponde à un behavior pattern λ_{free} [1] : il suffit alors de vérifier si le programme exhibe le comportement

1. Notons que le behavior pattern λ_{free} dépend du type de la donnée d. Si d est un identifiant de fichier, λ_{free} sera le behavior pattern décrivant la fermeture de fichier, si d

$M := \exists x.\, \lambda_1(x) \wedge \neg\lambda_{free}(x)\, \mathbf{U}\, \lambda_2(x)$. Ainsi, alors même que le comportement M était a priori inconnu (il n'était pas référencé dans une base de données de comportement suspects), notre méthode a permis de le découvrir automatiquement. Bien sûr, les comportements composés de fonctionnalités non corrélées (par exemple $M' := \exists x, y.\, \lambda_1(x) \odot \lambda_2(y)$) seront sans doute sans intérêt pour la plupart, mais les comportements ayant des actions corrélées (comme le comportement M précédent) seront susceptibles d'apporter des informations bien plus enrichissantes. Cette découverte de comportements peut bien évidemment s'appliquer à des comportements de grande taille.

Notons que ce scénario est rendu réellement possible par l'abstraction : c'est en identifiant les behavior patterns λ_1 et λ_2 dans l'ensemble des traces d'exécution que le comportement M a pu être découvert, ce que nous n'aurions pas pu faire sans l'abstraction. Par ailleurs, les comportements découverts représentent une information non triviale pour un analyste et l'efficacité et le cadre formel de nos techniques rendent ce scénario tout à fait réaliste. Enfin, les comportements découverts peuvent typiquement être utilisés pour compléter une base de comportements suspects existants : notre technique permet ainsi de suggérer de nouvelles signatures à l'analyste.

Classification des programmes malveillants. Les méthodes de classification reposent traditionnellement sur des techniques d'apprentissage automatique (cartes auto-adaptatives, machines à vecteurs de support, etc.). Ces techniques opèrent sur un espace vectoriel : chaque dimension de l'espace vectoriel correspond à une caractéristique du programme (est-il chiffré ? communique-t-il sur le réseau ? quel est le profil statistique de son code ? etc.). La classification des programmes malveillants repose ensuite sur un partitionnement de l'espace vectoriel, par identification (par apprentissage automatique) des zones où se concentrent des programmes malveillants. Ainsi, on pourra identifier une zone de l'espace où les programmes malveillants connus sont tous des keyloggers, etc. Ces méthodes sont décrites plus en détail dans [98] et dans [65]. Dans ce contexte, notre méthode peut fournir des caractéristiques supplémentaires (i.e. des dimensions supplémentaires dans l'espace vectoriel), afin d'affiner la classification. Il peut s'agir de caractéristiques discrètes (le programme exhibe tel comportement ou pas) ou de caractéristiques conti-

est une chaîne de caractères, λ_{free} sera le behavior pattern décrivant l'écrasement ou la libération de cette chaîne, etc. Le type de la donnée d est déterminé par son usage dans les patterns λ_1 et λ_2 : par exemple, si λ_1 désigne une ouverture de fichier, on saura que d est un identifiant de fichier et pas une chaîne de caractères.

nues (le programme exhibe tel comportement avec telle incertitude) qui décrivent par exemple l'exhibition d'un comportement d'envoi de mails, d'un comportement de persistance (inscription au redémarrage de l'ordinateur), d'un comportement de parcours des emplacements réseau, etc.

Analyse de similarité. Lorsqu'un programme n'est pas détecté comme exhibant le même comportement qu'un malware connu, on peut néanmoins vouloir étudier si son comportement est similaire (au moins partiellement) au comportement de programmes malveillants connus. Ainsi, dans le cas des termes (resp. dans le cas des mots), on peut calculer l'automate de traces partiellement abstrait (resp. Γ-abstrait) et comparer l'automate résultant à des automates partiellement abstraits (resp. Γ-abstraits) de référence. On peut également comparer l'ensemble de comportements découverts automatiquement par la méthode précédente à des ensembles de comportements de référence.

7.2 Perspectives

Nous avons vu que notre formalisme d'abstraction nous permettait de détecter des comportements complexes, par exemple de fuite d'information. En particulier, nous pouvions tenir compte de la transformation des données dérobées avant leur envoi sur le réseau. Cette transformation des données n'est pas propre à la fuite d'information, elle doit être prise en compte dans tout comportement manipulant des données. Or c'est surtout la dépendance de données que l'on suit : par exemple, on souhaite détecter un comportement $M := \exists x, y.\, \lambda_1(x) \odot \lambda_2(y)$ où le paramètre y est construit à partir du paramètre x. Nous l'avons pris en compte en bornant le nombre maximal de dépendances et en représentant alors explicitement les transformations. Ainsi, nous représentons le comportement par une formule exprimant tous les cas :

$$M := (\exists x.\, \lambda_1(x) \odot \lambda_2(x)) \vee (\exists x, y.\, \lambda_1(x) \odot \lambda_{depends}(y, x) \odot \lambda_2(y)) \vee \ldots$$

La question qui se pose est donc de savoir si on peut adapter notre formalisme pour décrire dans M, de manière concise, un nombre non borné de dépendances.

Une idée est d'utiliser le μ-calcul modal (avec données) pour représenter les comportements abstraits et les behavior patterns. Cette logique temporelle définit en particulier un opérateur de plus grand point fixe et elle est mise en œuvre dans l'outil CADP que nous avons vu au Chapitre 5. L'opérateur de plus grand point fixe, noté μ, permet alors de représenter un nombre quelconque

de transformations de données à l'aide de la formule de μ-calcul suivante (avec les notations de CADP) :

$$\exists x.\, \lambda_1(x) \odot \mu Y(y := x).\, (\lambda_2(y) \vee (\exists z.\, \lambda_{depends}(z, y) \odot Y(z))).$$

Cette formule décrit l'observation d'une action $\lambda_1(d)$ suivie, plus tard, soit d'une action $\lambda_2(d)$, soit d'une action $\lambda_{depends}(d', d)$ suivie, plus tard, soit d'une action $\lambda_2(d')$, soit d'une action $\lambda_{depends}(d'', d')$ etc. La vérification d'une telle formule a une complexité qui reste linéaire en la taille du modèle [82] et elle est mise en œuvre dans des outils existants, dont l'outil EVALUATOR4 de CADP.

Le problème est de savoir si un tel comportement a toujours la propriété de (m, n)-complétude, pour des entiers m et n, ou, si ce n'est pas le cas, s'il existe tout de même une procédure de détection décidable.

Plus généralement, certains comportements sont difficilement représentables dans notre formalisme. Par exemple, considérons le cas de la duplication de handles : le handle d'un fichier peut être dupliqué et les opérations sur ce fichier exécutées sur l'un quelconque des handles. Ainsi, on peut observer la trace (simplifiée) suivante :

```
h = CreateFile(...);
h′ = DuplicateHandle(h);
WriteFile(h, "...");
WriteFile(h′, "...");
WriteFile(h, "...");
```

En fait, la question qui se pose plus fondamentalement est de savoir si la dépendance des données peut être inscrite directement dans la représentation des traces, c'est-à-dire dans le formalisme préalable à l'opération d'abstraction.

Une autre perspective de ce travail est la définition d'un formalisme d'abstraction à plusieurs couches, au sens de Martignoni et al. [81], mais appliqué aux termes. Ce formalisme n'apporte pas plus d'expressivité dans la définition des comportements mais son approche étagée permet un degré d'abstraction plus fin et donc plus robuste qu'avec une seule couche. Là-encore, le problème de la décidabilité de la détection, à l'aide de propriétés similaires à la (m, n)-complétude, se pose.

Une dernière perspective de ce travail est son intégration au sein de méthodes de détection existantes. Par exemple, l'analyse morphologique [24] compare des graphes de flot de contrôle où un nœud représente non pas une instruction mais un bloc d'instructions, ce qui permet de se protéger contre des modifications simples du code. Le graphe du programme est alors comparé par

isomorphisme de sous-graphe aux graphes de programmes malicieux connus. En particulier, cette approche permet de détecter qu'une partie du graphe du programme est similaire à une partie du graphe d'un malware et ainsi de déduire que le programme analysé est probablement une variante de ce malware ou en a au moins repris une partie du code.

On pourrait envisager une approche similaire à un niveau comportemental : un nœud représenterait non pas une instruction ou un groupe d'instructions mais une fonctionnalité abstraite. Les graphes exploités cumuleraient alors une information morphologique et une information fonctionnelle. Ainsi, la robustesse apportée par l'abstraction de comportements serait mise au service de l'analyse morphologique.

Bibliographie

[1] Blade : ASU Proof Slicing Model Checker. http://asusrl.eas.asu.edu/blade.

[2] BLAST : Berkeley Lazy Abstraction Software Verification Tool. http://mtc.epfl.ch/software-tools/blast.

[3] CBMC : Model Checking for ANSI-C. http://www.cprover.org/cbmc.

[4] DynamoRIO. http://dynamorio.org.

[5] The IDA Pro Disassembler and Debugger. http://www.hex-rays.com/idapro.

[6] The Java PathFinder. http://javapathfinder.sourceforge.net.

[7] OpenFST. http://www.openfst.org/.

[8] Pin. http://www.pintool.org.

[9] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[10] Athanasios Alexandrakis and Symeon Bozapalidis. Weighted grammars and kleene's theorem. *Inf. Process. Lett.*, 24 :1–4, January 1987.

[11] Martin Apel, Christian Bockermann, and Michael Meier. Measuring similarity of malware behavior. In *IEEE Conference on Local Computer Networks*, pages 891–898. IEEE, October 2009.

[12] Gogul Balakrishnan, Radu Gruian, Thomas W. Reps, and Tim Teitelbaum. Codesurfer/x86-a platform for analyzing x86 executables. In Rastislav Bodík, editor, *Compiler Construction, 14th International Conference, CC 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3443 of *Lecture Notes in Computer Science*, pages 250–254. Springer, 2005.

[13] Thomas Ball, Ella Bounimova, and Leonardo de Moura. Efficient evaluation of pointer predicates with Z3 SMT solver in SLAM2. Technical Report MSR-TR-2010-24, Microsoft Research, 2010.

[14] Thomas Ball, Ella Bounimova, Rahul Kumar, and Vladimir Levin. Slam2 : Static driver verification with under 4 In *Proceedings of FMCAD 2010*, 2010.

[15] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 203–213, New York, NY, USA, 2001. ACM.

[16] Piotr Bania. Generic unpacking of self-modifying, aggressive, packed binary programs. *CoRR*, abs/0905.4581, 2009.

[17] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauscheck, Christopher Kruegel, and Engin Kirda. Scalable, Behavior-Based Malware Clustering. In *16th Symposium on Network and Distributed System Security (NDSS)*, 2009.

[18] Philippe Beaucamps and Jean-Yves Marion. On behavioral detection. In *Proceedings of EICAR'09*, May 2009.

[19] Ralf Benzmüller and Sabrina Berkenkop. Half-yearly malware report : July - december 2010. Technical report, G Data, 2011.

[20] J. Bergeron, M. Debbabi, J. Desharnais, MM. Erhioui, Y. Lavoie, and N. Tawbi. Static detection of malicious code in executable programs. In *Symposium on Requirements Engineering for Information Security*, 2001.

[21] Jean Berstel, Jr. and Christophe Reutenauer. *Rational series and their languages*. Springer-Verlag New York, Inc., New York, NY, USA, 1988.

[22] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast : Applications to software engineering. *Int. J. Softw. Tools Technol. Transf.*, 9 :505–525, October 2007.

[23] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Lazy shape analysis. In *CAV'06*, pages 532–546, 2006.

[24] Guillaume Bonfante, Matthieu Kaczmarek, and Jean-Yves Marion. Architecture of a morphological malware detector. *Journal in Computer Virology*, 2008.

[25] Ronald V. Book and Friedrich Otto. *String-rewriting systems*. Springer-Verlag, London, UK, 1993.

[26] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. Automatically identifying trigger-based behavior in malware. *Botnet Detection*, 36 :65–88, 2008.

[27] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Detecting self-mutating malware using control-flow graph matching. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, volume 4064 of *Lecture Notes in Computer Science*, pages 129–143. Springer, 2006.

[28] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. *SIGPLAN Not.*, 25 :296–310, June 1990.

[29] Feng Chen and Grigore Roşu. MOP : An Efficient and Generic Runtime Verification Framework. In *Object-Oriented Programming, Systems, Languages and Applications(OOPSLA'07)*, pages 569–588. ACM press, 2007.

[30] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. Semantics-aware malware detection. In *IEEE Symposium on Security and Privacy*, pages 32–46. IEEE Computer Society, 2005.

[31] Cristina Cifuentes and K. John Gough. Decompilation of binary programs. *Softw. Pract. Exper.*, 25 :811–829, July 1995.

[32] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[33] Fred Cohen. Computer viruses : Theory and experiments. *Computers and Security*, 6(1) :22–35, 1987.

[34] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on : http://www.grappa.univ-lille3.fr/tata, 2008.

[35] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera : extracting finite-state models from java source code. In *Proceedings of the 22nd*

international conference on Software engineering, ICSE '00, pages 439–448, New York, NY, USA, 2000. ACM.

[36] Claudio Demartini, Radu Iosif, and Riccardo Sisto. dSPIN : A Dynamic Extension of SPIN. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, pages 261–276, London, UK, 1999. Springer-Verlag.

[37] Christophe Devine and Nicolas Richaud. A study of anti-virus' response to unknown threats. In *Proceedings of EICAR'09*, May 2009.

[38] Thomas Dullien and Rolf Rolles. Graph-based comparison of executable objects. In *Symposium sur la Sécurité des Technologies de l'Information et des Télécommunications*, 2005.

[39] Matthew B. Dwyer, John Hatcliff, Matthew Hoosier, Venkatesh Prasad Ranganath, Robby, and Todd Wallentine. Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In *TACAS'06*, pages 73–89, 2006.

[40] Matthew B. Dwyer, John Hatcliff, Robby, and Venkatesh Prasad Ranganath. Exploiting object escape and locking information in partial-order reductions for concurrent object-oriented programs. *Formal Methods in System Design*, 25 :199–240, 2004. 10.1023/B :FORM.0000040028.49845.67.

[41] Joost Engelfriet, Zoltán Fülöp, and Heiko Vogler. Bottom-up and top-down tree series transformations. *J. Autom. Lang. Comb.*, 7 :11–70, July 2001.

[42] Javier Esparza, Peter Rossmanith, and Stefan Schwoon. A uniform framework for problems on context-free grammars. *Bulletin of the EATCS*, 72 :169–177, 2000.

[43] Andrea Flexeder, Bogdan Mihaila, Michael Petter, and Helmut Seidl. Interprocedural control flow reconstruction. In *Proceedings of the 8th Asian conference on Programming languages and systems*, APLAS'10, pages 188–203, Berlin, Heidelberg, 2010. Springer-Verlag.

[44] Zoltán Fülöp and Heiko Vogler. Weighted tree automata and tree transducers. In Manfred Droste, Werner Kuich, and Heiko Vogler, editors, *Handbook of Weighted Automata*, Monographs in Theoretical Computer Science. An EATCS Series, pages 313–403. Springer Berlin Heidelberg, 2009.

[45] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2006 : A Toolbox for the Construction and Analysis of Distributed Processes. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification (CAV'2007)*, volume 4590 of *Lecture Notes in Computer Science*, pages 158–163, Berlin Germany, 2007.

[46] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. Cadp 2010 : A toolbox for the construction and analysis of distributed processes. In Parosh Abdulla and K. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605 of *Lecture Notes in Computer Science*, pages 372–387. Springer Berlin / Heidelberg, 2011.

[47] Thomas Genet. *Reachability Analysis of Rewriting for Software Verification*. Habilitation à diriger les recherches, Université de Rennes 1, 2009.

[48] Rémi Gilleron and Sophie Tison. Regular Tree Languages and Rewrite Systems. *Fundamenta Informaticae*, 24 :157–176, 1995.

[49] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems : An Approach to the State-Explosion Problem*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[50] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart : directed automated random testing. *SIGPLAN Not.*, 40(6) :213–223, 2005.

[51] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. In *Network Distributed Security Symposium*. Internet Society, 2008.

[52] J. F. Groote and R. Mateescu. Verification of temporal properties of processes in a setting with data, 1998.

[53] Carl A. Gunter. *Semantics of Programming Languages : Structures and Techniques*. MIT Press, 1992.

[54] Hackerzvoice. Reversing android.

[55] John Hatcliff, Matthew B. Dwyer, Shawn Laubach, and A Muhammad. Specializing configurable systems for finite-state verification. Technical report, 1998.

[56] John Hatcliff, Matthew B. Dwyer, and Hongjun Zheng. Slicing software for model construction. *Higher-Order and Symbolic Computation*, 13 :315–353, 2000. 10.1023/A :1026599015809.

[57] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2 :366–381, 2000.

[58] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *J. Comput. Secur.*, 6 :151–180, August 1998.

[59] Gerard J. Holzmann. *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley Professional, sep 2003.

[60] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. *SIGPLAN Not.*, 24 :28–40, June 1989.

[61] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 35–46, New York, NY, USA, 1988. ACM.

[62] Liang Huang and David Chiang. Better k-best parsing. In *Proceedings of the Ninth International Workshop on Parsing Technology*, Parsing '05, pages 53–64, Stroudsburg, PA, USA, 2005. Association for Computational Linguistics.

[63] C. R. Iosif and R. Sisto. Modeling and validation of Java multi-threading applications using Spin. In *Proceedings of the 4th International SPIN Workshop*, 1998.

[64] Radu Iosif, Matthew B. Dwyer, and John Hatcliff. Translating java for multiple model checkers : The bandera back-end. *Formal Methods in System Design*, 26 :137–180, 2005. 10.1007/s10703-005-1491-3.

[65] Grégoire Jacob, Hervé Debar, and Eric Filiol. Behavioral detection of malware : from a survey towards an established taxonomy. *Journal in Computer Virology*, 4 :251–266, 2008.

[66] Grégoire Jacob, Hervé Debar, and Eric Filiol. Malware behavioral detection by attribute-automata using abstraction from platform and language. In *International Symposium on Recent Advances in Intrusion Detection*, volume 5758 of *Lecture Notes in Computer Science*, pages 81–100. Springer, 2009.

[67] Ke Jiang. Model Checking C Programs by Translating C to Promela. Master's thesis, Uppsala Universitet, 2009.

[68] Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of lisp-like structures. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '79, pages 244–256, New York, NY, USA, 1979. ACM.

[69] Neil D. Jones and Steven S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '82, pages 66–74, New York, NY, USA, 1982. ACM.

[70] Johannes Kinder, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. Detecting malicious code by model checking. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, volume 3548 of *Lecture Notes in Computer Science*, pages 174–187. Springer, 2005.

[71] Johannes Kinder, Helmut Veith, and Florian Zuleger. An abstract interpretation-based framework for control flow reconstruction from binaries. In Markus Müller-Olm Neil D. Jones, editor, *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2009)*, volume 5403 of *LNCS*, pages 214–228. Springer, 2009.

[72] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. Kemmerer. Behavior-based Spyware Detection. In *Proceedings of the 15th USENIX Security Symposium*, Vancouver, BC, Canada, August 2006.

[73] Kevin Knight and Jonathan May. Applications of weighted automata in natural language processing. In Manfred Droste, Werner Kuich, and Heiko Vogler, editors, *Handbook of Weighted Automata*, Monographs in Theoretical Computer Science. An EATCS Series, pages 571–596. Springer Berlin Heidelberg, 2009.

[74] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and XiaoFeng Wang. Effective and efficient malware detection at the end host. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 351–366, Berkeley, CA, USA, 2009. USENIX Association.

[75] Fred Kröger and Stephan Merz. *Temporal Logic and State Systems*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008.

[76] Werner Kuich. Tree transducers and formal tree series. *Acta Cybern.*, pages 135–164, 1999.

[77] Werner Kuich and Arto Salomaa. *Semirings, Automata and Languages.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1985.

[78] William Landi and Barbara G. Ryder. Pointer-induced aliasing : a problem taxonomy. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '91, pages 93–103, New York, NY, USA, 1991. ACM.

[79] Baudouin Le Charlier, Abdelaziz Mounji, and Morton Swimmer. Dynamic detection and classification of computer viruses using general behaviour patterns. In *International Virus Bulletin Conference*, pages 1–22, 1995.

[80] Mandiant. M-trends 2010 - the advanced persistent threat. Technical report, Mandiant, 2010.

[81] Lorenzo Martignoni, Elizabeth Stinson, Matt Fredrikson, Somesh Jha, and John C. Mitchell. A layered architecture for detecting malicious behaviors. In *International symposium on Recent Advances in Intrusion Detection*, volume 5230 of *Lecture Notes in Computer Science*, pages 78–97. Springer, 2008.

[82] Radu Mateescu and Damien Thivolle. A model checking language for concurrent value-passing systems. In *World Congress on Formal Methods*, pages 148–164, 2008.

[83] Jonathan May, Kevin Knight, and Heiko Vogler. Efficient inference through cascades of weighted tree transducers. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, ACL '10, pages 1058–1066, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.

[84] McAffee Labs. McAfee Threats Report : Fourth Quarter 2010. Technical report, McAffee, 2011.

[85] Kenneth McMillan. Lazy annotation for program testing and verification. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 104–118. Springer Berlin / Heidelberg, 2010.

[86] Mehryar Mohri. Finite-state transducers in language and speech processing. *Comput. Linguist.*, 23 :269–311, June 1997.

[87] Mehryar Mohri. Statistical natural language processing. In *Applied Combinatorics on Words*, chapter 4. Cambridge University Press, New York, NY, USA, 2005.

[88] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *IEEE Symposium on Security and Privacy*, pages 231–245. IEEE Computer Society, 2007.

[89] Younghee Park, Douglas Reeves, Vikram Mulukutla, and Balaji Sundaravel. Fast malware classification by automated behavioral graph matching. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, CSIIRW '10, pages 45 :1–45 :4, New York, NY, USA, 2010. ACM.

[90] Danny Quist and Valsmith. Covert debugging circumventing software armoring techniques. In *Blackhat Briefings*, 2007.

[91] William C. Rounds. Mappings and grammars on trees. *Theory of Computing Systems*, 4 :257–287, 1970.

[92] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and Wenke Lee. Polyunpack : Automating the hidden-code extraction of unpack-executing malware. In *Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual*, pages 289 –300, dec. 2006.

[93] Barbara G. Ryder, William A. Landi, Philip A. Stocks, Sean Zhang, and Rita Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Trans. Program. Lang. Syst.*, 23 :105–186, March 2001.

[94] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 16–31, New York, NY, USA, 1996. ACM.

[95] SAnTos Laboratory. The Bandera tool set for model checking concurrent Java software. http://bandera.projects.cis.ksu.edu.

[96] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *IEEE Symposium on Security and Privacy*, pages 144–155. IEEE Computer Society, 2001.

[97] SGDN / DCSSI / SDO / BC. Menaces sur les systèmes informatiques. Technical report, Direction Centrale de la Sécurité des Systèmes d'Information, France, 2006.

[98] Muazzam Ahmed Siddiqui. *Data mining methods for malware detection*. PhD thesis, Orlando, FL, USA, 2008. AAI3335368.

[99] Prabhat K. Singh and Arun Lakhotia. Static verification of worm and virus behavior in binary executables using model checking. In *Information Assurance Workshop*, pages 298–300. IEEE Press, 2003.

[100] James W. Thatcher. Generalized2 sequential machine maps. *Journal of Computer and System Sciences*, 4(4) :339 – 367, 1970.

[101] Sean Thompson. A survey on model checking java programs, 1999.

[102] Ferucio Laurentiu Tiplea and Erkki Mäkinen. On the complexity of a problem on monadic string rewriting systems. *Journal of Automata, Languages and Combinatorics*, 7(4) :599–609, 2002.

[103] Giovanni Vigna. Static disassembly and code analysis. In Mihai Christodorescu, Somesh Jha, Douglas Maughan, Dawn Song, and Cliff Wang, editors, *Malware Detection*, volume 27 of *Advances in Information Security*, pages 19–41. Springer US, 2007.

Nancy-Université
INPL

AUTORISATION DE SOUTENANCE DE THESE
DU DOCTORAT DE L'INSTITUT NATIONAL
POLYTECHNIQUE DE LORRAINE

o0o

VU LES RAPPORTS ETABLIS PAR :
Madame Sandrine BLAZY, Professeur, IRISA, RENNES

Madame Sophie TISON, Professeur, Université des Sciences et Technologies de Lille, Villeneuve d'Ascq

Le Président de l'Institut National Polytechnique de Lorraine, autorise :

Monsieur BEAUCAMPS Philippe

à soutenir devant un jury de l'INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE,
une thèse intitulée :

"Analyse de Programmes Malveillants par Abstraction de Comportements. "

en vue de l'obtention du titre de :

DOCTEUR DE L'INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE

Intitulé du doctorat : **« Informatique »**

Fait à Vandoeuvre, le 27 octobre 2011
Le Président de l'I.N.P.L.,
F. LAURENT

INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE – NANCY

2 av. de la Forêt de Haye | BP 3
54501 Vandœuvre-lès-Nancy Cedex
FRANCE
Tél. : +33(0)3 83 59 59 59
Fax : +33(0)3 83 59 59 55

Résumé

L'analyse comportementale traditionnelle opère en général au niveau de l'implantation de comportements malveillants. Pourtant, elle s'intéresse surtout à l'identification de fonctionnalités données et elle se situe donc plus naturellement à un niveau fonctionnel. Dans cette thèse, nous définissons une forme d'analyse comportementale de programmes qui opère non pas sur les interactions élémentaires d'un programme avec le système mais sur la fonction que le programme réalise. Cette fonction est extraite des traces d'un programme, un procédé que nous appelons abstraction. Nous définissons de façon simple, intuitive et formelle les fonctionnalités de base à abstraire et les comportements à détecter, puis nous proposons un mécanisme d'abstraction applicable à un cadre d'analyse statique ou dynamique, avec des algorithmes pratiques à complexité raisonnable, enfin nous décrivons une technique d'analyse comportementale intégrant ce mécanisme d'abstraction. Notre méthode est particulièrement adaptée à l'analyse des programmes dans des langages de haut niveau ou dont le code source est connu, pour lesquels l'analyse statique est facilitée : applications mobiles en .NET ou Java, scripts, extensions de navigateurs, composants off-the-shelf.

Le formalisme d'analyse comportementale par abstraction que nous proposons repose sur la théorie de la réécriture de mots et de termes, les langages réguliers de mots et de termes et le model checking. Il permet d'identifier efficacement des fonctionnalités dans des traces et ainsi d'obtenir une représentation des traces à un niveau fonctionnel ; il définit les fonctionnalités et les comportements de façon naturelle, à l'aide de formules de logique temporelle, ce qui garantit leur simplicité et leur flexibilité et permet l'utilisation de techniques de model checking pour la détection de ces comportements ; il opère sur un ensemble quelconque de traces d'exécution ; il prend en compte le flux de données dans les traces d'exécution ; et il permet, sans perte d'efficacité, de tenir compte de l'incertitude dans l'identification des fonctionnalités. Un cadre d'expérimentation a été mis en place dans un contexte d'analyse dynamique comme statique.

Mots clés : détection de codes malveillants, analyse comportementale, abstraction de comportements, model checking, réécriture de mots, réécriture de termes, analyse statique, automates à états finis.

Abstract

Traditional behavior analysis usually operates at the implementation level of malicious behaviors. Yet, it is mostly concerned with the identification of given functionalities and is therefore more naturally defined at a functional level. In this thesis, we define a form of program behavior analysis which operates on the function realized by a program rather than on its elementary interactions with the system. This function is extracted from program traces, a process we call abstraction. We define in a simple, intuitive and formal way the basic functionalities to abstract and the behaviors to detect, then we propose an abstraction mechanism applicable both to a static or to a dynamic analysis setting, with practical algorithms of reasonable complexity, finally we describe a behavior analysis technique integrating this abstraction mechanism. Our method is particularly suited to the analysis of programs written in high level languages or with a known source code, for which static analysis is facilitated: mobile applications for .NET or Java, scripts, browser addons, off-the-shelf components.

The formalism we propose for behavior analysis by abstraction relies on the theory of string and terms rewriting, word and tree languages and model checking. It allows an efficient identification of functionalities in traces and thus the construction of a representation of traces at a functional level; it defines functionalities and behaviors in a natural way, using temporal logic formulas, which assure their simplicity and their flexibility and enables the use of model checking techniques for behavior detection; it operates on an unrestricted set of execution traces; it handles the data flow in execution traces; and it allows the consideration of uncertainty in the identification of functionalities, with no complexity overhead. Experiments have been conducted in a dynamic and static analysis setting.

Keywords: malware detection, behavioral analysis, behavior abstraction, model checking, string rewriting, term rewriting, static analysis, finite state automata.