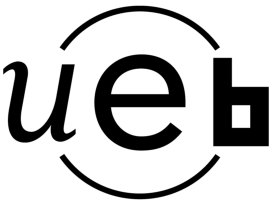
Génération et évaluation de mécanismes de détection des intrusions au niveau applicatif

Jonathan-Christofer Demay

N° d'ordre : 4268 ANNÉE 2011



THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : INFORMATIQUE

École doctorale Matisse

présentée par

Jonathan-Christofer Demay

préparée à l'unité de recherche EA 4039 (SSIR)
Sécurité des Systèmes d'Information et des Réseaux
SUPELEC - École Supérieure d'Électricité

Génération et évaluation de mécanismes de détection des intrusions au niveau applicatif

Thèse soutenue à Rennes le 1 juillet 2011

devant le jury composé de :

César VIHO
Irisa / président
Hervé DEBAR
Télécom SudParis / rapporteur
Vincent NICOMETTE
Insa Toulouse / rapporteur
Benjamin MONATE
CEA - LIST / examinateur
Ludovic MÉ
SUPELEC / directeur de thèse
Éric TOTEL
SUPELEC / co-directeur de thèse
Frédéric TRONEL
SUPELEC / co-directeur de thèse

Remerciements

Cette thèse a été réalisée au sein de l'équipe Sécurité des Systèmes d'Information et Réseaux (SSIR) du campus de Rennes de SUPELEC grâce à un financement de l'Agence Nationale de la Recherche (ANR).

Hervé Debar et Vincent Nicomette ont accepté la tâche de rapporteur, je les remercie pour la rapidité avec laquelle ils ont lu mon manuscrit et pour le contenu de leur rapport. Je remercie César Viho, pour avoir accepté le rôle de président du jury et Benjamin Monate, pour avoir accepté celui d'examinateur.

Je tiens bien sûr à remercier Éric Totel pour m'avoir encadré depuis le stage de master jusqu'à la préparation de la soutenance, mais aussi Frédéric Tronel, qui a pris le train en marche pour devenir co-encadrant et faire de même. À l'issue de ces travaux de thèse, je peux dire que j'ai beaucoup appris en travaillant à leur côté durant toutes ces années.

Je remercie également Ludovic Mé pour, entre autre, son cours de master sur la sécurité informatique, celui-là même qui m'a donné envie de tenter l'aventure de la thèse à Rennes.

Un grand merci aussi à Frédéric Majorczyk et à Alain Degardin qui connaissent maintenant grâce à moi, eux aussi, la joie d'écrire des scénarios d'exécution et de réaliser des campagnes de tests. Vous fûtes tous deux d'une aide précieuse pour l'obtention de résultats expérimentaux. Tous mes remerciements aussi à Pascal Cuoq pour ses explications à propos du fonctionnement interne de *Frama-C*.

Merci également à tous les membres de l'équipe SSIR ainsi qu'à toutes les personnes que j'ai pu cotoyer au laboratoire pendant ces années de thèse. Je ne pense pas trop m'avancer en disant qu'il ne sera pas facile de retrouver ailleurs une ambiance de travail aussi chaleureuse et détendue.

Merci aussi à mes parents qui m'ont permis de continuer les études aussi longtemps que je le souhaitais et sans qui cette thèse de doctorat n'aurait pas été possible.

Enfin, je remercie la ville de Rennes et ses habitants, où il est décidément bien agréable de vivre, à tel point qu'on y reste plus longtemps que prévu ☺.

Table des matières

Table des figures

Liste des tableaux

Introduction

Les systèmes d'information mais aussi Internet, le réseau mondial qui interconnecte nombre d'entre eux, jouent un rôle grandissant dans la vie quotidienne et dans notre société en général. En effet, des domaines relevant de la vie privée tels que l'envoi de courrier ou bien le paiement à distance, mais aussi des domaines stratégiques comme le secteur bancaire ou encore les communications militaires, reposent de nos jours massivement sur les systèmes d'information. De ce fait, des attaques réalisées par des utilisateurs malveillants et visant à exploiter les vulnérabilités de ces systèmes d'information sont de plus en plus fréquentes. De telles attaques peuvent par exemple nuire à l'image du propriétaire du système d'information ou causer d'importants dommages financiers. La problématique de la sécurité devient donc une question essentielle aussi bien pour les utilisateurs que pour les administrateurs de ces systèmes d'information.

Tout appareil destiné à traiter de manière automatique de l'information peut être qualifié de système informatique. Les systèmes d'information reposent sur un ensemble organisé de systèmes informatiques et de système de télécommunications pour gérer de l'information dans un contexte précis. Cette gestion comprend par exemple les tâches d'élaboration, de modification, de stockage ou encore de transport de l'information. Ces systèmes sont par nature en proie à des menaces vis-à-vis de l'information qu'ils traitent. En effet, pour un système d'information donné, un utilisateur malveillant pourrait chercher à altérer ou à détruire des données (on parle alors de perte d'intégrité de l'information), à révéler illégitimement des données à un tiers (on parle alors de perte de confidentialité de l'information), ou encore à empêcher un accès légitime à des données (on parle alors de perte de disponibilité de l'information).

Pour répondre à la problématique de la sécurité, une politique doit être définie en fonction du système d'information que l'on souhaite sécuriser et des objectifs de sécurité que l'on souhaite atteindre. Cette politique exprime les propriétés de confidentialité, d'intégrité et de disponibilité que doit respecter le système d'information

afin de garantir sa sécurité. Pour faire respecter ces propriétés, des mécanismes préventifs, notamment de contrôle d'accès, sont mis en œuvre sur les systèmes d'information. Ces mécanismes ont accès à tout ou partie de la politique de sécurité et devraient être capables d'empêcher de manière préventive toute action qui aboutirait à une violation d'une des propriétés qu'elle exprime.

Cependant, la conception des systèmes d'information et de leurs différents composants est une tâche complexe. Cela implique que le risque qu'une faute de conception ou de configuration soit présente dans le système d'information n'est pas nul. Ces fautes sont autant de vulnérabilités potentielles que pourraient chercher à exploiter un utilisateur malveillant pour contourner les mécanismes préventifs et effectuer des opérations proscrites par la politique de sécurité. On désigne alors par le terme intrusion toute violation intentionnelle d'une des propriétés exprimées par la politique de sécurité.

Tenant compte de l'éventualité d'une intrusion malgré la présence de mécanismes préventifs, des mécanismes capables de détecter une violation de la politique de sécurité une fois que celle-ci a effectivement eu lieu ont été mis en place. On appelle ce type de mécanismes des systèmes de détection d'intrusion. L'objectif de ces mécanismes et de lever une alerte en cas d'intrusion afin de la signaler à l'administrateur du système. Ce dernier devra alors procéder à l'analyse du rapport d'alerte pour remettre le système d'information dans un état opérationnel mais aussi pour identifier la vulnérabilité à l'origine de l'intrusion afin de pouvoir la corriger. Eventuellement, l'administrateur pourra également juger de la sévérité de la compromission. Par exemple, une compromission qui a permis à l'utilisateur malveillant d'obtenir sur le système d'information ciblé des droits administrateur est plus sévère que si des droits restreints ont seulement pu être obtenus.

Les systèmes de détection d'intrusion peuvent être classés en deux catégories : ceux qui cherchent à détecter des malveillances (on parle alors d'approches par signature) et ceux qui cherchent à détecter des anomalies (on parle alors d'approches comportementale). Dans le premier cas, le modèle de détection repose sur la connaissance que l'on a des attaques tandis que dans le second cas, celui-ci repose sur la connaissance que l'on a de l'entité surveillée en situation de fonctionnement normal. Une approche comportementale présente l'avantage de pouvoir détecter des attaques encore inconnues au moment de la modélisation. Toutefois, la construction d'un tel modèle de détection peut être une tâche difficile. En effet, il n'est pas simple de définir ce qui est caractéristique du comportement normal de l'entité surveillé et toute erreur de modélisation risque d'entraîner la levée de fausses alertes. Le modèle de détection d'intrusion que nous proposons dans cette thèse est de type comportemental.

Au niveau logiciel, on distingue parmi les vulnérabilités à l'origine des intrusions les fautes de conception préalablement présentes dans les programmes et les fautes de configuration de ces derniers. Les attaques qui exploitent les fautes de conception peuvent également être classées en deux catégories : les attaques contre les données

de contrôle et les attaques contre les données de calcul. Les données de contrôle sont utilisées par le programme pour gérer son flot d'exécution tandis que les données de calcul sont utilisées pour contenir la valeur des variables présentes dans le code source du programme. Dans le premier cas, une attaque va chercher à faire dévier illégalement le flot d'exécution du programme vers des instructions invalides (par exemple, vers du code injecté). Dans le second cas, elle va chercher à modifier le flux d'information du programme pour utiliser de manière illégale des instructions valides (par exemple, au travers d'un chemin incorrect). Les travaux que nous présentons dans cette thèse s'inscrivent dans le domaine de la détection d'intrusion au niveau logiciel. Nous ciblons donc les intrusions qui exploitent les vulnérabilités présentes dans les applications. Plus précisément, nous ciblons les intrusions qui exploitent les vulnérabilités présentes dans les applications pour réaliser des attaques contre les données de calcul.

Il se trouve que ce type d'attaque est capable de mettre en échec une grande partie des systèmes de détection d'intrusion proposés jusqu'à maintenant. Pour les rares d'entre eux qui seraient capable de les détecter, le mécanisme de détection implique une surcharge à l'exécution qui est élevée et donc un coût important pour la mise en production. C'est pourquoi nous proposons, dans ces travaux de thèse, un modèle de détection d'intrusion qui se concentre en particulier sur ce type bien précis d'attaque, et ce avec pour objectif d'obtenir une surcharge à l'exécution qui soit plus faible que celle des autres mécanismes auxquels il peut se comparer. De plus, afin de tester notre modèle de détection dans un contexte où les vulnérabilités et les différentes manières de les exploiter ne sont pas connues à l'avance, nous proposons aussi un modèle d'injection de fautes pour évaluer les systèmes de détection d'intrusion face aux attaques contre les données de calcul.

Pour comprendre notre démarche vis-à-vis des intrusions qui résultent d'une attaque contre les données de calcul, nous devons tout d'abord caractériser de manière générale une intrusion du point de vue de la sûreté de fonctionnement. Pour un système d'information donné et la politique de sécurité qui lui est associée, une intrusion est une défaillance de ce système. En effet, une intrusion est le résultat d'une erreur de fonctionnement au sein de ce système et cette erreur est elle-même le résultat d'une faute activée par un utilisateur malveillant. La faute à l'origine d'une intrusion n'est donc possible que par la présence d'une vulnérabilité dans le système. Pour détecter une attaque contre les données de calcul, nous allons construire un modèle de comportement normal qui permet de détecter au sein des applications les erreurs engendrées par ce type d'attaque. Pour évaluer notre mécanisme de détection, nous allons construire un modèle de faute qui permet de simuler, par injection, l'état erroné dans lequel se trouve le programme après avoir été la cible d'une attaque contre les données de calcul.

La construction d'un modèle de comportement normal, y compris au niveau applicatif, peut être réalisée de deux manières différentes : par analyse du comportement de l'entité surveillée durant une phase d'apprentissage (on considère alors que

le programme est dans un état de fonctionnement normal, c'est-à-dire sans présence d'intrusion) ou bien par analyse du programme lui-même ou de sa spécification (on parle alors de modèles construits statiquement). De plus, les modèles construits statiquement peuvent être classés en trois catégories suivant que la source de l'analyse permette une connaissance nulle, une connaissance partielle ou bien une connaissance complète du fonctionnement interne de l'entité surveillée. Il s'agit respectivement des méthodes dîtes en boîte noire, en boîte grise et en boîte blanche. La méthode de construction que nous proposons dans ces travaux de thèse afin d'obtenir notre modèle de détection pour un logiciel donné repose sur l'analyse de son code source. Il s'agit donc d'une méthode construite statiquement par une analyse en boîte blanche de l'entité surveillée.

La construction d'un modèle de faute peut avoir deux objectifs : assister la découverte de fautes de conception et automatiser l'évaluation des mécanismes de détection ou de tolérance aux fautes. Tout d'abord, le cas où le modèle de faute permet d'assister la découverte de fautes de conception. Il s'agit ici de mettre en évidence et ce de manière automatisée des fautes de conception que pourraient déclencher intentionnellement ou non un utilisateur. Dans cette situation, le mécanisme d'injection se place au même niveau qu'un utilisateur normal du système. Le programme ne peut alors être mis dans un état erroné que si celui-ci contient bien au moins une faute de conception et que si le modèle de faute permet bien de générer au moins un scénario d'exécution capable d'en déclencher une. Si le programme à tester est un service réseau et si on admet l'hypothèse que l'utilisateur, malveillant ou non, utilise nécessairement un client distant, alors la connaissance du protocole de communication peut être utilisée pour générer automatiquement de nombreux comportements potentiellement malveillants.

Ensuite, le cas où le modèle de faute permet d'évaluer des mécanismes de détection ou de tolérance aux fautes. Il s'agit ici de mettre le programme dans un état interne erroné afin de tester les capacités de détection ou de tolérance des mécanismes à évaluer. Dans cette situation, le mécanisme d'injection se place au même niveau que le système d'exploitation. Le mécanisme d'injection a par exemple ainsi la capacité de modifier directement l'espace mémoire des processus. Le programme peut alors être mis dans un état interne erroné sans avoir besoin d'activer une faute de conception existante. Ce que nous proposons dans ces travaux de thèse pour évaluer les détecteurs d'intrusion logiciels vis-à-vis des attaques contre les données de calcul, c'est d'utiliser le second type de mécanismes d'injection pour simuler (c'est-à-dire sans connaissance *a priori* de vulnérabilités existantes) ce type particulier d'attaques.

Dans ces travaux de thèse, nos contributions sont les suivantes :

- Un modèle de détection d'intrusion ciblant les attaques contre les données de calcul et présentant un très faible surcoût à l'exécution ainsi qu'un taux de faux positif nul.

- Un modèle d'injection de fautes reproduisant le résultat d'une attaque contre les données de calcul et permettant d'évaluer les mécanismes de détection face à ce type particulier d'attaque.
- Une implémentation de ces modèles de détection et d'injection fondés sur un analyseur statique de code dédié au programme écrit en langage *C* et utilisable sur des programmes réels.
- Un protocole automatisé d'évaluation des détecteurs d'intrusion de niveau applicatif dans le cas d'une application réseau et reposant sur notre modèle d'injection de fautes.

Ces contributions ont donné lieu aux publications suivantes :

- *SIDAN : a tool dedicated to Software Instrumentation for Detecting Attacks on Non-control-data*, publié à *CRISIS 2009* [DTT09a] présente en détails notre modèle de détection ainsi que l'outil que nous avons développé. Cet article correspond aux travaux abordés dans les chapitres 2 et 4.
- *Automatic Software Instrumentation for the Detection of Non-control-data Attacks*, publié à *RAID 2009* [DTT09b], est une version préliminaire des travaux précédemment cités.
- *Detecting illegal system calls using a data-oriented detection model*, publié à *IFIPSEC 2011* [DMTT11], présente notre modèle de simulation d'attaque ainsi que la plateforme d'évaluation que nous avons développé. Cet article correspond aux travaux abordés dans les chapitres 3 et 4.
- *Génération et évaluation de mécanismes de détection d'intrusion au niveau applicatif*, publié à SAR-SSI 2010 [DTT10], est une version initiale en français des travaux précédemment cités.

Le mémoire est organisé de la manière suivante :

Le premier chapitre est consacré à l'état de l'art. Il présente les travaux antérieurs par rapport auxquels situer nos contributions mais aussi les domaines liés à l'approche que nous proposons dans ces travaux de thèse. Nous commençons par aborder les travaux antérieurs par rapport auxquels nous situons notre approche pour la détection. La première section de ce chapitre est consacrée au domaine de la détection d'intrusion hôte ainsi qu'aux différentes approches utilisées dans ce domaine. L'approche pour la détection que nous proposons dans ces travaux de thèse repose sur un modèle comportemental au niveau applicatif. La seconde section aborde donc plus en détails ce type de modélisation. Nous abordons ensuite les travaux issus de la sûreté de fonctionnement sur lesquels reposent nos approches. Notre approche pour la détection d'intrusion repose sur un modèle de détection d'erreurs au sein des logiciels, nous présentons donc ce domaine dans la troisième section. Notre approche pour l'évaluation repose sur un modèle d'injection de fautes, nous présentons enfin ce domaine dans la quatrième et dernière section.

Le second chapitre est consacré à la première partie importante de notre travail : la détection d'intrusion. Il présente d'abord les catégories d'attaques qui peuvent cibler les applications et comment les informations contenues dans le code source des programmes peuvent nous permettre de les détecter. Pour cela, nous commençons par aborder dans la première section les différents types de données que peut cibler une attaque contre un programme et pour chacun d'entre eux la manière de corrompre ces données pour réaliser une intrusion. Puis, nous expliquons dans la deuxième section comment il est possible à partir de certaines informations contenues dans le code source de détecter une catégorie d'attaques bien précise, les attaques contre les données de calcul. Le second chapitre présente ensuite comment il est possible de construire de manière entièrement automatisée un modèle de détection sur ce principe et comment nous avons implémenté ce type de modèle dans le cas des programmes écrits en langage *C*. Pour cela, nous commençons par expliquer dans la troisième section que notre modèle de détection est orienté autour des variables dont dépendent les appels de fonctions et quelles techniques d'analyse statique peuvent être utilisées pour le construire. Puis, nous présentons dans la quatrième section l'outil que nous avons développé pour tester notre approche et comment nous avons répondu aux problèmes posés par la construction du modèle de détection dans le cas particulier de notre implémentation.

Le troisième chapitre est consacré à la seconde partie importante de ces travaux de thèse : l'évaluation des mécanismes de détection des intrusions vis-à-vis des attaques contre les données de calcul. Dans la première section, nous commençons par aborder la problématique de la simulation de l'état interne erronée d'un programme suite à une attaque de ce type. Pour cela, nous présentons d'abord le modèle de faute que nous proposons afin d'arriver à ce résultat. Puis, nous discutons de l'utilisation du modèle ainsi proposé dans le cadre de l'évaluation du taux de détection pour un programme donné. Nous abordons ensuite dans la seconde section le mécanisme d'injection de fautes que nous avons implémenté pour appliquer ce modèle de faute dans le cas des programmes écrits en langage *C*. Pour cela, nous expliquons d'abord que nous avons choisi d'avoir à nouveau recours à l'analyse statique de code et à l'instrumentation. Puis, nous présentons comment ce mécanisme d'injection doit être utilisé dans le cadre d'une campagne de tests.

Le quatrième chapitre est consacré aux résultats de l'évaluation de notre mécanisme de détection d'intrusion ciblant les attaques contre les données de calcul à l'aide du mécanisme d'injection de fautes que nous avons proposé au chapitre précédent. Dans la première section, nous commençons par présenter la plateforme de tests que nous avons développée. Pour cela, nous abordons tout d'abord la problématique de la génération de scénarios d'exécution qui permettent pour un programme donné d'obtenir un taux de couverture de code acceptable. Puis, nous présentons les différentes informations qui doivent nécessairement être collectées par la plateforme de tests afin de réaliser l'évaluation des mécanismes de détection suite à une campagne d'injections. Dans la deuxième section, nous présentons ensuite les résultats de l'é-

valuation de notre approche pour la détection. Pour cela, nous détaillons d'abord les performances du mécanisme d'analyse et de génération des invariants ainsi que l'impact de l'instrumentation sur l'exécution des programmes ainsi analysés. Puis, nous détaillons les performances de la détection suite à l'utilisation de la plateforme de tests ainsi que l'utilisation, à titre de comparaison, d'un autre mécanisme de détection basé sur *libanomaly* [KMVV03, KMRV03].

Le dernier chapitre conclut cette thèse par un récapitulatif des travaux que nous avons menés et des résultats que nous avons obtenus. Pour finir, nous donnons quelques perspectives pour des travaux futurs dans la continuité de ces travaux de thèse.

Chapitre 1

État de l'art

Cette thèse est liée aux domaines de la détection d'intrusion hôte, des modèles de détection comportementaux, de la détection d'erreur et de l'injection de fautes. En effet, nous y proposons un mécanisme de détection d'intrusion hôte basé sur un modèle comportemental et reposant sur la détection d'erreur au sein des programmes. Nous proposons également de l'évaluer à l'aide d'un mécanisme d'injection de fautes, et ce de manière entièrement automatisée. Ce chapitre présente les travaux de recherche antérieurs issus de ces domaines.

La première section de ce chapitre porte sur la détection d'intrusion hôte. Nous présentons d'abord brièvement la différence entre les systèmes de détection réseaux et les systèmes de détection hôtes. Puis, le mécanisme de détection que nous proposons relevant de la seconde catégorie, nous abordons ensuite dans le cadre des mécanismes hôtes les deux grandes approches pour la détection, à savoir les modèles de détection par signature et les modèles de détection comportementaux. Puis, pour chacun d'entre eux nous présentons les différentes méthodes qui ont été proposées pour construire ces modèles.

La seconde section de ce chapitre se concentre sur les modèles comportementaux au niveau applicatif. En effet, le modèle de détection d'intrusion que nous proposons dans cette thèse, de type comportemental, cible uniquement et individuellement les applications. Nous abordons alors pour ce type d'approche des travaux antérieurs suivant le niveau de connaissance qu'ils possèdent sur le fonctionnement interne des applications. Dans chacun des cas, nous présentons les différentes méthodes proposées pour construire et vérifier ces modèles.

La troisième section de ce chapitre aborde la problématique de la détection d'erreur. En effet, le mécanisme de détection d'intrusion que nous proposons dans cette thèse est inspiré d'un certain nombre de travaux antérieurs issus de ce domaine. Tout d'abord, nous présentons brièvement la notion de sûreté de fonctionnement

avant d'aborder les notions de base que sont les notions de faute, d'erreur et de défaillance. Par la suite, nous présentons les mécanismes permettant de détecter ou de tolérer des erreurs au sein des applications. Plus précisément, le système de détection d'intrusion que nous proposons repose sur un mécanisme de niveau logiciel et embarqué dans les programmes. Ce type particulier de mécanisme et donc abordé plus en détails dans cette dernière partie.

La quatrième et dernière section de ce chapitre porte sur l'injection de fautes. Les mécanismes d'injection de fautes ont été initialement utilisés pour assister la découverte d'erreurs de conception et dans l'évaluation des mécanismes de détection et de tolérance aux erreurs d'exécution. Nous présentons d'abord le fonctionnement de la procédure d'injection puis les différents modèles utilisés pour générer les données qui lui sont nécessaires. En effet, pour évaluer le système de détection d'intrusion que nous proposons, nous voulons simuler des attaques contre les applications. Pour cela, la méthode que nous proposons repose sur un mécanisme d'injection de fautes au niveau logiciel.

1.1 La détection d'intrusion hôte

La politique de sécurité d'un système d'information définit les propriétés de confidentialité, d'intégrité et de disponibilité que ce dernier doit respecter. Le respect de la confidentialité implique que seules les personnes autorisées ont accès en lecture aux éléments considérés. De même, le respect de l'intégrité implique que seules les personnes autorisées ont accès en écriture aux éléments considérés et que ces derniers sont modifiés de manière correcte. Enfin, le respect de la disponibilité implique que ces éléments considérés puissent être accessibles par les personnes autorisées au moment voulu. Une attaque contre un système d'information a pour but de violer ces propriétés. Si une telle attaque réussit alors l'intrusion est effective. Une intrusion dans un système d'information peut donc être défini comme une violation de sa politique de sécurité [And80].

Les systèmes de détection d'intrusion sont des outils permettant aux administrateurs de systèmes d'information d'être alertés en cas d'intrusion ou même de tentative d'intrusion afin de pouvoir réagir de manière adéquate. Certains outils, nommés systèmes de réaction, permettent d'aller plus loin dans l'automatisme [Tho07]. En effet, en plus de lever une alerte lorsqu'une intrusion est détectée, ces mécanismes prennent des mesures pouvant aboutir au blocage de l'intrusion (exemple : bannir l'adresse *IP* de l'attaquant).

Les mécanismes de détection d'intrusion reposent sur l'utilisation de sondes. Celles-ci ont la capacité de collecter des informations qui, par une analyse ultérieure vont permettre de mettre en évidence la présence d'une attaque connue ou l'absence de comportement normal (voir section 1.1.2). Ces sondes peuvent être naturellement présentes sur le système d'information (par exemple, les différents fichiers journaux

maintenus par le système d'exploitation ou par les applications). Elles peuvent aussi être fournies par le mécanisme de détection (par exemple, un renifleur de paquets réseaux ou encore un module spécifique à charger par le système d'exploitation ou par une application).

En fonction du placement de ces sondes, on distingue deux grandes catégories de système de détection d'intrusion : les systèmes de détection réseaux et les systèmes de détection hôtes. Dans le premier cas, les sondes sont placées au niveau des équipement réseaux et collectent les informations qui transitent sur le réseau. Dans le second cas, les sondes sont placées sur les machines qui exploitent le réseau et collectent les informations qui transitent au niveau du système d'exploitation ou au niveau des applications. Suivant le type de système de détection d'intrusion utilisé, il est plus ou moins facile de détecter certaines intrusions (par exemple, une attaque de type déni de service par *syn flooding* est plus facilement détectable au niveau réseau [Edd07]).

Dans les deux cas, les informations collectées par les sondes peuvent être utilisées par le mécanisme de détection de deux manières différentes : pour détecter des malveillances ou pour détecter des anomalies. Il s'agit respectivement des approches par signature et des approches comportementales. Le premier type d'approche repose sur la connaissance que l'on a des attaques tandis que dans le second sur la connaissance que l'on a du comportement de l'entité surveillée en situation de fonctionnement normal. Dans cette thèse, c'est un mécanisme de détection hôte que nous proposons. Nous allons donc maintenant présenter plus en détails ces deux approches dans le cadre de la détection hôte.

1.1.1 Modèles de détection par signature

Les détecteurs d'intrusion par signature reposent sur la création *a priori* d'une base de motifs représentant des scénarios d'attaque connus au préalable. Cette base de signature est ensuite utilisée, le plus souvent en temps réel, sur les informations fournies par les sondes de détection. C'est un système de reconnaissance de motifs qui permet de mettre en évidence dans ces informations la présence d'une intrusion connue de la base de signature.

Ces mécanismes de reconnaissance sont souvent peu consommateur de ressources et la pertinence de la détection de ce type d'approche est élevée. C'est pour ces deux raisons que ce type de système est souvent choisi pour être utilisé en production [PREL]. Cette démarche est similaire à celle employée par les mécanismes de détection des programmes malveillants. Par exemple, l'outil *Hancock* [GSHC09] propose d'extraire automatiquement à partir d'un ensemble de programmes malveillants des séquences d'octets ayant une très faible probabilité d'apparaître dans d'autres programmes.

Dans le cas des intrusions, les signatures sont créées à l'aide de langage de description d'attaque [lLWJ98, PD01, EVK02]. Leur création relève le plus souvent

d'une tâche manuelle. Toutefois, des méthodes ont été proposées pour générer ces signatures de manière automatique [NS05, MG00]. Ces deux approches sont présentées plus en détails dans la suite de cette section. Le taux de couverture de ce type d'approche repose principalement sur la complétude de la base de signature ainsi que sur la qualité des motifs qui y sont contenus. En effet, le plus souvent une signature est associée à une attaque particulière. De ce fait, seules les attaques dont la signature est présente dans la base sont détectées. Cette approche présente donc deux inconvénients majeurs.

D'une part, elle ne permet pas de détecter des attaques inconnues. Cela signifie qu'entre le moment où une nouvelle attaque est conçue et le moment où celle-ci est connue, le système d'information est vulnérable malgré la présence d'un mécanisme de détection. Notons qu'il est possible de chercher à généraliser les signatures afin qu'il ne suffise pas de légèrement modifier une attaque pour réussir à contourner le mécanisme de détection. Toutefois, cette généralisation des signatures risque de faire augmenter le taux de faux positifs [PD00]. Notons aussi qu'une autre manière de généraliser une signature n'est pas de décrire l'exploitation de la vulnérabilité mais la vulnérabilité elle même [BNS+06].

D'autre part, la base de signature doit être régulièrement mise à jour. Comme pour les mécanismes de détection des programmes malveillants, cette tâche est consommatrice en ressources. En effet, cela nécessite de la part des personnes chargées de la maintenance de la base de veiller en permanence à l'apparition de nouvelles attaques puis de prendre le temps d'écrire les signatures correspondantes. Cette fois, cela signifie qu'entre le moment où une nouvelle attaque est connue et le moment où la base est mise à jour, là encore le système d'information est vulnérable malgré la présence d'un mécanisme de détection.

1.1.1.1 Langages de description d'attaques

L'approche par signature se base sur la connaissance *a priori* du mode opératoire des attaques et par conséquent sur la connaissance d'activités caractéristiques à chacune de ces attaques. Ces activités caractéristiques se traduisent sur le système ciblé par des évènements observables eux aussi potentiellement caractéristiques de ces attaques. Pour décrire précisément une attaque, il faut pouvoir décrire ces différents évènements observables. Pour cela, il est nécessaire de disposer de langages adaptés à cette tâche.

Des langages ont été proposés avec pour objectif de se concentrer sur l'exploitation des failles. Ils permettent de décrire une à une les étapes nécessaires à l'exploitation d'une faille particulière [VEK00, EVK00]. Certains de ces langages ont d'ailleurs été proposés originalement pour automatiser l'exploitation des failles dans le cadre des tests d'intrusions [Sec98, Der99]. Cependant, il est important de pouvoir préciser l'état du système avant la réalisation de l'attaque. En effet, pour être réalisées, certaines attaques nécessitent qu'un certain nombre de conditions soit réu-

nies (type de matériel ciblé, version du système d'exploitation, version de l'application, etc.). Des langages permettent explicitement d'exprimer ce genre de conditions [LMPT98, CO00, MM01].

D'autre langages choisissent de se concentrer sur les évènements observables au niveau du système. Par exemple, c'est notamment le cas des signatures des projets *ORCHIDS* [OGl05, GO08] et *GnG* [TVM04]. Des évènements tels que les appels système ou l'activation de certaines fonctions du noyau sont utilisés pour décrire les attaques. Ces évènements sont utilisés pour créer des automates décrivant les états et les transitions du système qui permettent de reconnaître la présence d'une intrusion en cours de réalisation. De plus, pour les évènements qui possèdent des paramètres, il est possible d'utiliser des opérateurs sur ces paramètres pour simplifier la description de l'attaque mais aussi pour exprimer des contraintes complexes sur ces derniers [PD00].

1.1.1.2 Découverte de signature d'attaques

Des travaux ont cherché à découvrir de manière automatique des signatures d'attaques [MG00]. Ces derniers se basent sur l'exploration de données. L'objectif de ces travaux est de réussir à caractériser un type de comportement bien précis pour les applications. Concrètement, cette approche repose sur l'utilisation d'une machine à états finis pour apprendre le type de comportement d'un groupe d'applications similaires dans le but de pouvoir ensuite reconnaître si une application donnée appartient ou non à ce groupe.

Par exemple, en effectuant la phase d'apprentissage sur un ensemble de traces d'exécution de différents navigateurs *web*, l'automate est ensuite capable de reconnaître la présence d'un navigateur *web* (qui ne faisait pas partie de la phase d'apprentissage) parmi un ensemble quelconque de traces d'exécution. En utilisant cette méthode d'apprentissage sur un ensemble de traces d'exécution d'applications différentes mais toutes compromises par un même type d'attaque, l'automate est alors capable de détecter sur d'autres applications une attaque similaire bien que celle-ci soit encore inconnue au moment de l'apprentissage.

1.1.2 Modèles de détection comportementaux

Les détecteurs d'intrusion comportementaux reposent sur la création d'un modèle de référence représentant le comportement de l'entité surveillée en situation de fonctionnement normal. Ce modèle est ensuite utilisé durant la phase de détection afin de pouvoir mettre en évidence d'éventuelles déviations comportementales. Pour cela, le comportement de l'entité surveillée est comparé à son modèle de référence. Une alerte est levée lorsqu'une déviation trop importante (notion de seuil) vis-à-vis de ce modèle de comportement normal est détectée. Le principe de cette approche est de considérer tout comportement n'appartenant pas au modèle de comportement

normal comme une anomalie symptomatique d'une intrusion ou d'une tentative d'intrusion. Deux caractéristiques permettent alors de qualifier la précision d'un modèle de détection : son caractère correct et son caractère complet (voir figure 1.1).

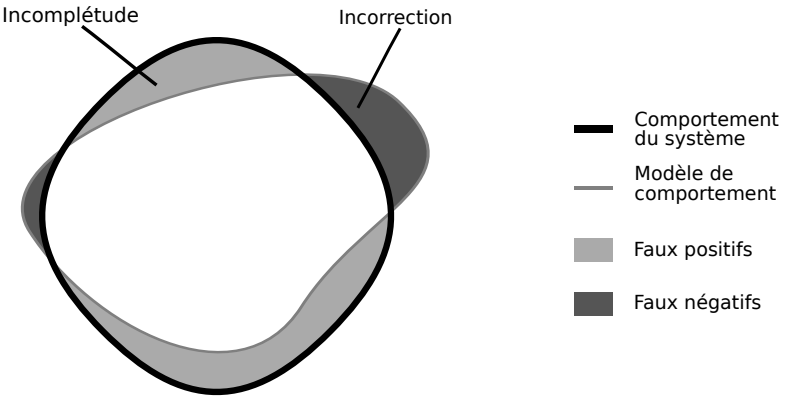


FIGURE 1.1 – Précision d'un modèle de comportement normal

Nous définissons comme correct un modèle de comportement normal si celui-ci modélise, du point de vue de la politique de sécurité, tout ou partie du comportement légitime de l'entité surveillée. Un tel modèle détecte toutes les intrusions et ne présente pas de faux-négatif (absence d'alerte en présence d'attaques). Toutefois, bien que correct, le modèle de comportement normal peut contenir une part de sous-approximation. Dans ce cas, celui-ci peut par contre présenter des faux-positifs (levée d'alertes sans présence d'attaques).

De manière duale, nous définissons comme complet un modèle de comportement normal si celui-ci modélise, du point de vue de la politique de sécurité, l'intégralité du comportement de l'entité surveillée. Un tel modèle ne détecte que des intrusions et ne présente pas de faux-positif. Toutefois, bien que complet, le modèle de comportement normal peut contenir une part de sur-approximation. Dans ce cas, celui-ci peut par contre présenter des faux-négatifs. Pour évaluer un modèle de détection, il faut donc évaluer son caractère correct et son caractère complet. Bien souvent, seule une estimation est obtenue à partir d'une mesure de son taux de faux positifs et de son taux de faux négatifs lors d'une ou plusieurs campagnes de tests.

Le principal avantage des approches comportementales par rapport aux approches par signature est que celles-ci permettent de détecter des attaques encore inconnues au moment de la modélisation. En effet, le modèle de détection est construit non pas dans le but de caractériser les attaques à l'origine des intrusions mais dans le but de caractériser les déviations comportementales engendrées par les intrusions. De ce fait, une intrusion inconnue au moment de la création du modèle de référence est tout de même détectée tant que celle-ci produit une déviation observable par rapport au modèle de référence.

Une approche de type comportementale est donc caractérisée par la méthode employée pour construire le modèle de référence mais aussi par celle utilisée pour évaluer la différence entre le comportement de référence et celui observé durant la phase de détection. Le plus souvent, ces deux points sont liés : l'évaluation de la déviation est adaptée au type de modélisation sur lequel repose le mécanisme de détection. La précision du comportement de référence, c'est-à-dire sa propension à présenter des faux-négatifs ou à émettre des faux-positifs, va donc dépendre de la méthode choisie pour la phase de construction du modèle.

Dans les travaux antérieurs, plusieurs approches ont été proposées pour construire ce modèle de référence. On peut les classer suivant que celles-ci reposent sur une phase d'apprentissage (durant laquelle on suppose que le système ne subit pas d'attaque) ou sur une définition préalable du modèle de comportement (une spécification par exemple).

Dans le premier cas, il s'agit d'observer l'entité surveillée durant une période où l'on considère qu'il n'y a pas d'attaque. Le modèle est ainsi construit sur la base des informations récoltées durant la phase d'observation. Ce type d'approche présente le risque que les informations ainsi obtenues ne permettent pas de modéliser complètement le comportement de l'entité surveillée. En effet, il existe un risque que la phase d'observation n'ait pas permis de couvrir l'ensemble des comportements attendus de l'entité surveillée. Pour pallier ce problème, certaines méthodes de ce type proposent soit un mécanisme de modélisation capable de généraliser soit un mécanisme de détection capable de tolérer un seuil de déviation avant de lever une alerte. Cela diminue effectivement le risque de faux positifs mais en contre-partie augmente celui de faux-négatifs. De plus, la détermination du seuil de généralisation ou du seuil de détection vient alors s'ajouter à la difficulté de configuration du mécanisme de détection considéré.

Dans le second cas, il s'agit d'analyser une description formelle du comportement de l'entité surveillé. Celle-ci peut être obtenue par exemple à partir de la spécification du système, de son code source ou de sa configuration. La difficulté de ce type d'approche repose dans la capacité à extraire de manière automatique les informations nécessaires à la construction du modèle de référence. En effet, la difficulté de construction du modèle peut venir de la complexité d'analyse de la la description formelle (c'est par exemple le cas du code source) ou du fait que celle-ci n'est pas toujours complète (cela peut arriver avec un fichier de spécification ou de configuration). Nous présentons maintenant ces deux types d'approches pour la construction du modèle de détection.

1.1.2.1 Méthodes construite par une phase d'apprentissage

Les méthodes par apprentissage reposent sur une phase d'observation de l'entité surveillée. Les données récoltées durant la phase d'observation sont ensuite utilisées pour construire le modèle de comportement normal de l'entité considérée. Pour cela,

l'hypothèse est faite qu'aucune intrusion ou tentative d'intrusion n'a lieu pendant la phase d'observation. Afin d'exploiter durant la phase de détection les données ainsi récoltées durant la phase d'observation, plusieurs approches ont été proposées pour caractériser à partir de ces données récoltées le comportement normal de l'entité surveillée. Dans la suite de cette section ces approches sont présentées dans le cadre de la détection d'intrusion hôte.

Système expert De manière générale, un système expert est un mécanisme capable de répondre à une question sur un ensemble de faits. L'objectif de ce type d'outil est de simuler le raisonnement d'un expert du domaine auquel se rapporte la question. Pour cela, le mécanisme travaille à partir d'un ensemble de règles spécifiquement écrites pour une question donnée. Ces règles sont ensuite vérifiées sur le jeu de faits fourni à l'aide d'un moteur d'inférence. Répondre à la question revient donc à vérifier si le jeu de faits fourni respecte ou non les règles. Notons qu'il est possible de déduire automatiquement un jeu de règles, pour une question donnée, à partir d'un jeu de faits initial.

Ce type d'approche peut être utilisé pour analyser automatiquement de l'information. Il peut par exemple servir à classifier les comportements chez les utilisateurs d'un système informatique [VL89]. Dans ces travaux, les sondes du système informatique sont utilisées pour collecter un ensemble d'informations sur les comportements des utilisateurs normaux. Puis, l'ensemble de ces informations est utilisé comme jeu de donnée initial pour la déduction des règles. Enfin, durant la phase de détection, le système expert ainsi obtenu est utilisé sur les informations fournies par les sondes du système informatique. Ce dernier permet ainsi de détecter les comportements inhabituels d'un utilisateur alors jugé comme malveillant.

Modèle statistique Un modèle statistique peut être utilisé pour décrire mathématiquement un mécanisme observé. Généralement, les observations ne permettent que d'obtenir une description approximative. Pour cela, la valeur de certaines observations sont considérées comme des variables aléatoires. Pour chacune de ses observations, un modèle statistique est utilisé pour décrire l'ensemble de distributions de la variable aléatoire correspondante.

Si on applique cette approche aux observations effectuées par les sondes d'un système informatique durant une période de fonctionnement supposé normal, il est possible d'établir un modèle de comportement normal de ce système. Une fois le modèle de référence ainsi établi, on a la capacité de détecter une anomalie en mesurant la distance entre ce modèle et de nouvelles observations provenant des sondes du système. Cette approche statistique a été appliquée notamment à la détection de comportements anormaux chez les utilisateurs d'un système informatique [And80, LTG$^+$90]. Ces mécanismes de détection ciblent des paramètres du système aussi variés que les tentatives d'authentification, la durée des sessions, le temps processeur utilisé, etc.

Algorithme génétique Il s'agit d'algorithmes évolutionnistes dont le but est d'obtenir, pour un problème qui ne possède pas de méthode exacte, une solution approchée. Ce type d'algorithme repose sur la notion de sélection naturelle. Pour cela, une population initiale est générée aléatoirement. Puis, pour chaque génération, une métrique permet de déterminer quel sous-ensemble de la population actuelle sera utilisé pour engendrer la génération suivante. La solution approchée est obtenue par améliorations successives.

Ce type de mécanisme a été utilisé pour apprendre le comportement des utilisateurs d'un système informatique [BR01]. L'utilisation d'algorithmes génétiques dans ce contexte permet de prédire le comportement d'un utilisateur à partir d'un ensemble de ses actions passées. Par exemple, dans [BR01], l'historique des commandes exécutées par l'utilisateur est utilisé pour caractériser son comportement passé.

Réseau de neurones artificiel De manière générale, les réseaux de neurones artificiels permettent de traiter de l'information selon une modélisation mathématique qui s'inspire du fonctionnement des réseaux de neurones biologiques. Ils sont composés de modèles simplifiés du neurone biologique appelés neurones formels. Ces derniers sont conçus comme des automates dotés d'une fonction de transfert qui calcule une valeur de sortie en fonction de ses entrées. Ils sont organisés en couche et s'interconnectent selon une topologie variable. Les paramètres des fonctions de transfert ainsi que la topologie d'interconnexion sont évolutifs, ils sont modifiables selon un procédé d'apprentissage.

Ce type de modélisation peut être utilisé entre autre pour classifier automatiquement de l'information. En effet, si le jeu de données utilisé pour l'apprentissage est suffisamment grand, ces derniers présentent une capacité de généralisation. Cela nécessite toutefois de posséder *a priori* des informations déjà classifiées pour la phase d'apprentissage. De plus, si le réseau de neurones artificiels présente plusieurs entrées ou bien des connexions récurrentes, celui-ci a la capacité de traiter les informations en tant que série temporelle. Ce type de mécanisme a justement été utilisé pour modéliser le comportement des utilisateurs d'un système informatique [DBS92]. Après un entraînement par les utilisateurs normaux du système, le réseau de neurones artificiels est capable de détecter la présence d'un utilisateur malveillant.

Approche immunologique Le système immunologique est un mécanisme biologique qui permet de faire la différence entre ce qui est normal (le soi) et ce qui ne l'est pas (le non-soi). Pour cela, ce mécanisme repose sur la reconnaissance de ce qui est normal (le soi). Il s'agit donc bien par analogie d'une approche comportementale. La phase d'apprentissage de ce mécanisme biologique se fait en le confrontant à un large échantillon d'éléments entrant dans la composition du soi. Du point de vue de la détection d'intrusion, cela revient à générer à partir des informations renvoyées par les sondes d'observation une base des séquences d'évènements à reconnaître comme normaux.

Durant la phase de détection, les séquences d'évènements observés sont comparées à cette base de référence. Toute séquence étrangère à cet ensemble est considérée comme anormale et donc comme une potentielle exploitation d'une vulnérabilité. La capacité de généralisation de cette approche repose dans le fait que même si l'ensemble des comportements possibles de l'entité surveillée n'ont pas été observés par les sondes durant la phase d'apprentissage, les séquences d'évènements générées à partir de ces derniers couvrent l'ensemble des séquences possibles. Cette méthode a été utilisée sur les séquences de commandes exécutées par un utilisateur sur un système informatique afin de déterminer si ce dernier est malveillant ou non [LB97b, LB97c, LB97a]. Dans ce cas, la base de référence est constituée de séquences de taille fixe générées à partir de l'historique des commandes. Ce type d'approche est étudié plus en détails dans la section 1.2.1.2 où nous présentons un mécanisme de niveau applicatif fonctionnant sur ce principe avec les séquences d'appels système.

Graphe de contrôle Un graphe est un ensemble de nœuds liés entre eux par des arcs. Un graphe peut être utilisé pour modéliser le comportement d'un système à partir du moment où celui-ci peut être décrit à l'aide d'un automate. Les nœuds du graphe représentent alors les différents états possibles du système et les arcs l'ensemble des transitions autorisées entre ces états. Le graphe est ensuite utilisé durant la phase de détection pour contrôler que le programme est toujours dans un état valide et que celui a été atteint au travers d'un chemin autorisé.

Ce type d'approche a par exemple été appliquée sur des journaux conservant l'ensemble des connexions à un système d'information particulier [NC03]. Dans ces travaux il s'agit d'organiser sous forme de graphes les données contenues dans les journaux de connexions. Par exemple, dans le cadre de connexions à un service de banque en ligne, le montant des transactions pour un utilisateur donné, le nombre de ses transactions pour une période de temps donnée ou le type de transaction sont utilisés pour déterminer des motifs de graphes représentant le comportement d'un utilisateur normal. Les graphes qui s'éloignent le plus des graphes de référence sont alors considérés comme des anomalies et une alerte est levée par le système de détection d'intrusion.

Ce type d'approche a également été appliquée dans le cadre des systèmes diversifiés [MTMS07, MTMS08]. Dans ces travaux, les graphes de flux d'information produits par plusieurs serveurs *web* différents mais répondant à une même requête sont comparés. La difficulté de cette approche réside dans la comparaison des graphes. En effet, il faut être capable de masquer le non-déterminisme propre à ce type d'information mais aussi être capable de masquer les différences entres les différents serveurs *web*. Par exemple, la séquence d'appel système système nécessaire à l'envoi des données aux clients peut différer entre les serveurs *web* si ces derniers s'exécutent sur des systèmes d'exploitation différents.

De manière générale, ce type de mécanisme présente l'avantage de traiter efficace-

ment des évènements qui se produisent rarement sur le système mais qui néanmoins sont valides vis-à-vis du comportement de l'entité surveillée. De nombreuses autres modélisations comportementales qui reposent sur l'utilisation d'un graphe de contrôle sont mises en œuvre par des mécanismes de détection de niveau applicatif. Ce type bien particulier de système de détection d'intrusion hôte est abordé par la suite en détails dans la section 1.2.

1.1.2.2 Méthodes paramétrées par la politique de sécurité

La politique de sécurité comprend bien sûr la spécification du système auquel elle est associée. Au niveau hôte, celle-ci comprend également la politique de contrôle d'accès. Ce contrôle d'accès a pour objectif de garantir deux des propriétés de la sûreté de fonctionnement : la confidentialité et l'intégrité. Concrètement, lorsqu'une ressource est sollicitée, c'est cette politique qui en autorise ou non l'accès au sujet du système qui en a fait la demande. Une politique de contrôle d'accès implique des restrictions sur les flux d'informations : les informations contenues par un objet sont accessibles aux sujets du système qui peuvent lire cette ressource et celle dernière peut être modifiée par les sujets du système qui peuvent écrire dedans afin de propager des informations. Un mécanisme de détection d'intrusion peut donc utiliser cette politique de contrôle d'accès pour lever une alerte lorsqu'un flux d'information illégal est observé. Plusieurs familles de modèles de contrôle d'accès existent, nous les présentons maintenant ainsi que les mécanismes de détection associés.

Le contrôle d'accès discrétionnaire Ce type de contrôle d'accès permet de restreindre l'accès aux objets en fonction du sujet ou du groupe auxquels ils appartiennent. L'aspect discrétionnaire s'explique par le fait que c'est le sujet qui gère les contrôles d'accès de l'objet en fonction des droits qu'il possède sur l'objet en question. Concrètement, ce sont les utilisateurs du système qui attribuent les permissions sur les ressources dont ils sont propriétaires. La gestion de la politique de contrôle d'accès est donc déléguée aux propriétaires des ressources du système. Par exemple, pour les fichiers, ce sont les utilisateurs et les groupes propriétaires. La principale limitation de ce type de modèle réside dans la difficulté à l'administrer. En effet, si la granularité est grossière, l'administration sera simple mais il ne sera pas possible de gérer précisément la politique de contrôle d'accès. De même, si la granularité est fine, la gestion de la politique de contrôle d'accès sera précise mais l'administration sera une tâche complexe. Sur les systèmes *UNIX*, la gestion des permissions fonctionne sur un modèle grossier. Toutefois, des listes de contrôle d'accès peuvent être utilisées pour permettre une gestion plus fine que le système traditionnel de permissions.

Dans les deux cas, il s'agit de spécifier quels utilisateurs peuvent avoir accès aux objets ainsi que les différentes manipulations autorisées pour chacun d'eux. Une autorisation est exprimée par un couple contenant un sujet et une opération. Un exemple qui peut être exprimé dans les deux cas : si un fichier est associé au couple

(*user1*, *read*), alors l'utilisateur *user1* à l'autorisation de lire le contenu du fichier.

Blare est un *IDS* paramétré par la politique de contrôle d'accès d'un système *Linux* [ZMB02, ZMB03a, ZMB03b, GTM09]. Celui-ci interprète les droits d'accès exprimés par la politique de contrôle d'accès pour en déduire des restrictions sur les flux d'information. Pour cela, *Blare* repose sur la différenciation entre les transferts d'information directs et indirects. Un transfert d'information réalisé par une seule entité est un transfert direct. Un transfert d'information nécessitant la collaboration de plusieurs entités est un transfert indirect. Un transfert indirect d'information est jugé dangereux si un transfert direct équivalent n'est pas autorisé.

Pour illustrer cela, prenons comme exemple un système qui possède deux utilisateurs, a et b. Soit x, y et z trois fichiers de ce même système. Posons que a est autorisé à accéder en lecture au fichier x et en écriture au fichier y. Alors, le transfert du contenu initial du fichier x dans le fichier y est un flux d'information direct explicitement autorisé par la politique de contrôle d'accès. De même, posons que b est autorisé à accéder en lecture au fichier y et en écriture au fichier z. Alors, le transfert du contenu initial du fichier y dans le fichier z est un flux d'information direct explicitement autorisé par la politique de contrôle d'accès. Cependant, en collaborant, a et b peuvent transférer le contenu initial du fichier x dans le fichier z par l'intermédiaire du fichier y.

La politique de contrôle d'accès permet d'effectuer cette opération mais celle-ci n'autorise pas explicitement ce flux d'information. Cette opération potentiellement dangereuse est détectée par *Blare*.

Le contrôle d'accès obligatoire Il impose que les restrictions soient spécifiées par l'administrateur du système considéré et non par les utilisateurs. Ce type de modèle a d'abord été proposé pour répondre à des problématiques précises telle que la confidentialité pour l'accès à des documents classifiés [BL73, BL76]. Toutefois, un modèle de ce type a été proposé afin de pouvoir spécifier une politique de sécurité dans le cas général. Il s'agit du modèle *DTE* (pour *Domain and Type Enforcement*) [BK85]. En effet, le langage de spécification fourni par *DTE* est suffisamment général pour être utilisé pour de nombreuses problématiques. En contrepartie, cette généralité implique que le travail de spécification peut être fastidieux.

Le contrôle d'accès de *SELinux* [SELIN] implémente sous la forme d'un module de sécurité pour le noyau *linux* [LSM] un modèle de type *DTE*. Ce dernier permet de faire du confinement de processus [WSB+96] (donc de faire de la prévention) mais peut aussi être utilisé pour simplement lever des alertes (donc pour faire de la détection). La politique de sécurité est alors exprimée en spécifiant pour chaque programme (ou pour un sous-ensemble de ceux-ci) les actions autorisées. Les propriétés exprimées par la politique de sécurité vont dont permettre de décrire les restrictions d'accès aux ressources du système qui s'appliquent aux processus en cours d'exécution. Par exemple, dans le cas d'un serveur *web* s'exécutant sous le compte *root*, on peut restreindre son accès en lecture aux seuls fichiers contenant les pages *web*, et

ce malgré un niveau de privilèges permettant, via le contrôle d'accès discrétionnaire, de lire et de modifier l'ensemble des fichiers du système.

1.1.3 Positionnement des travaux

L'approche que nous proposons dans ces travaux est une approche de niveau hôte reposant sur un modèle comportemental. Ce que nous voulons avec cette approche, c'est détecter des attaques encore inconnues au moment de la création de notre modèle de comportement normal, et ce sans connaissance *a priori* des vulnérabilités exploitées par ces attaques. Plus précisément, ce sont les applications que nous ciblons et l'objectif de notre approche est de se concentrer sur un type bien particulier d'attaques, les attaques contre les données de calcul (voir section 2.1). Pour arriver à un résultat de détection qui ne présente jamais de faux positifs, nous utilisons un modèle construit statiquement et non un modèle construit par apprentissage. De plus, contrairement aux approches reposant sur une politique de contrôle d'accès, qui imposent une construction manuelle du modèle, nous voulons que notre modèle de détection soit construit de manière automatique. Pour cela nous avons choisi de nous reposer sur l'analyse statique du code source des programmes auxquels nous allons appliquer notre méthode de détection.

1.2 Les modèles comportementaux applicatifs

Au sein d'un programme, les informations sont contenues dans des données en mémoire. Ces dernières peuvent être classées suivant quelles contiennent des informations utilisées par le programme pour contrôler son flot d'exécution (par exemple, une adresse de retour sur la pile) ou pour effectuer ses calculs (par exemple, une valeur nécessaire à l'évaluation d'un branchement conditionnel). On parle respectivement de données de contrôle et de données de calcul. Ceci est abordé plus en détails à la section 2.1 à propos des attaques contre les applications.

Un mécanisme de détection d'intrusion peut adopter différents points de vue afin de récolter les informations nécessaires pour caractériser le comportement d'une application. Ceci est vrai quelque soit leur conteneur, données de contrôle ou bien données de calcul, ou encore la manière dont celles-ci sont récoltées, par une phase d'apprentissage ou bien par une analyse statique. Ce point de vue peut être complètement extérieur, à savoir que les processus sont considérés comme des boîtes noires et que seules les informations accessibles au niveau de l'interface entre les applications et le système d'exploitation sont utilisées pour caractériser le comportement. Ces modèles ne sont que peu efficaces contre les attaques visant les données de calcul et il est possible d'utiliser des techniques de contournement [KKMR05, PSJ07] sur les attaques visant les données de contrôle pour mettre en échec ce type de mécanisme de détection.

Des approches dîtes boîtes grises ont été proposées pour remédier à ce problème. Le mécanisme de détection possède un point de vue toujours principalement extérieur, néanmoins il accède à un certain nombre d'informations provenant de l'intérieur du processus et qui sont facilement accessibles de l'extérieur, notamment les structures de données utilisées par les appels système ou même les appels vers des bibliothèques de fonctions. Ces modèles améliorent la détection en terme de précision et de complétude puisqu'ils permettent de détecter des attaques contre les données de calcul et rendent plus difficile l'utilisation de techniques de contournement sur les attaques contre les données de contrôle. Toutefois, ces modèles n'ayant pas accès à l'ensemble des données internes des processus, ils sont toujours susceptibles de produire des fausses alertes ou de manquer des intrusions.

Pour pouvoir encore améliorer la détection il faut pouvoir aller encore plus loin dans la connaissance du fonctionnement interne des processus. C'est-à-dire qu'il faut adopter cette fois un point de vue intérieur et ainsi avoir accès à l'ensemble des algorithmes et des structures de données internes utilisées par l'application. C'est ce qu'on appelle les approches dîtes boîtes blanches. Ces approches ont la particularité de se concentrer respectivement sur un type bien précis d'attaque, soit contre les données de contrôle, soit contre les données de calcul. Ces trois perspectives d'approches sont détaillées dans la suite de cette section.

1.2.1 Les approches de type boîte noire

Les principales données accessibles au niveau de l'interface entre les processus et le système d'exploitation est la nature des appels système exécutés. Une approche qui surveille uniquement ce type d'information, et ce quelque soit le modèle de détection employé (voir section 1.1.2.1), est donc une approche de type boîte noire. Un utilisateur malveillant est capable de modifier l'enchaînement des appels système soit par une attaque contre les données de contrôle (par exemple en injectant du code) soit par une attaque contre les données calcul (par exemple en exécutant du code valide au travers d'un chemin incorrect). Une approche de type boîte noire ne pourra donc détecter une intrusion, et ce quelque soit le type des données attaquées, que si la déviation comportementale est visible au niveau des appels système. Ce type d'approche est donc vulnérable aux attaques par mimétisme [KKMR05, PSJ07]. Nous allons maintenant présenter plusieurs méthodes de modélisation comportementale qui ne reposent que sur la connaissance des appels système exécutés par les programmes.

1.2.1.1 Réseau de neurones artificiel

L'approche qui consiste à utiliser un réseau de neurones artificiels pour apprendre le comportement de l'entité surveillée (voir section 1.1.2.1 pour son application aux comportements utilisateurs d'un système informatique) peut être utilisée au niveau applicatif pour modéliser le comportement des applications [GSS99b, GSS99a]. Dans

ces travaux, les informations utilisées pour la détection sont les appels système effectués par les processus. Ces informations sont fournies par un ensemble de traces des précédentes exécutions des programmes obtenus grâce aux outils d'audit du système tel que *BSM* (*Basic Security Module*) sur *Solaris*. Les appels système ainsi obtenus sont fournis au réseau de neurones artificiels ainsi que l'ordre dans lequel ces derniers ont été observés.

Plus précisément, c'est un réseau de neurones artificiels de type récurrent qui est utilisé dans ces travaux. Un réseau de ce type possède des interconnections créant au moins un cycle dans le graphe représentant le réseau de neurones. Cela permet ainsi au réseau de pouvoir traiter l'information de manière non-linéaire. L'utilisation d'un réseau récurrent permet alors de tenir compte de l'ordre des informations dans l'apprentissage du comportement. De ce fait, les propriétés de généralisation des réseaux de neurones artificiels permettent ainsi de reconnaître des séquences d'appels système valides bien que ces dernières ne soient pas présentes dans les données d'apprentissage.

Durant la phase de détection, le réseau de neurones artificiels va fournir pour chaque séquence qu'il reçoit en entrée une mesure de l'anormalité de cette dernière. Ces mesures sont accumulées au fil des séquences jusqu'à ce qu'un certain seuil soit dépassé auquel cas une alerte est levée. Afin de pouvoir utiliser cette approche sur de longues exécutions des programmes, le cumul des mesures d'anormalité est régulièrement rabaissé. Le taux d'abaissement est configurable. Cette approche a été testée sur le jeu de données d'évaluation des systèmes de détection d'intrusion fourni par le *DARPA*. Avec un taux d'abaissement qui permet de ne produire aucun faux positif, le taux de détection est de 77.3%. Avec un taux d'abaissement plus faible et donc un taux de faux positifs non-négligeable, il est possible d'atteindre un taux de détection de 100%.

1.2.1.2 Approche immunologique

Le contrôle des séquences d'appels système effectuées par les processus est une approche simple initialement proposée par Forrest et al. [FHSL96, HFS98]. Dans ces travaux, le comportement normal d'un processus est caractérisé par un ensemble de traces d'exécution enregistrées durant la phase d'apprentissage. Ces traces contiennent de manière ordonnée l'ensemble des appels système effectués par le processus durant toute la durée de son exécution. Durant la phase de détection, les n derniers appels système effectués par le processus sont comparés à l'ensemble des sous-séquences de n éléments contenues dans les traces caractérisant le comportement de référence de ce processus. En pratique, la détection se fait à partir de la liste des séquences d'appels système de taille fixe n construites statiquement à partir des traces de référence.

La capacité de détection de ce mécanisme repose sur le choix du paramètre n. En effet, prenons pour exemple les deux situations extrêmes : lorsque le paramètre

n vaut 1 et lorsque celui-ci vaut ∞ (c'est-à-dire lorsque la séquence à comparer contient tous les appels systèmes effectués par le processus depuis son lancement). Dans le premier cas, si l'ensemble des appels système effectués par le processus lors d'une intrusion existent dans les traces de référence, alors le mécanisme de détection présentera un faux négatif en cas d'intrusion. Dans le second cas, si les traces de référence ne couvrent pas l'ensemble des comportements possibles du processus sans intrusion, alors le mécanisme de détection présentera un faux positif. La difficulté de configuration de ce type de mécanisme réside donc dans le choix de la valeur du paramètre n.

Wespi et al. [WDD00] ont proposé d'améliorer cette approche par l'utilisation d'une liste contenant des séquences d'appels système de taille variable. Dans ces travaux, la liste des séquences d'appels système est extraite des traces de référence à l'aide d'un algorithme de découverte de motif. Cette approche permet d'une part d'inclure dans le modèle de référence, des séquences d'appels système parfois très longues, mais néanmoins caractéristiques du comportement normal du processus quelle que soit la situation hors intrusion. D'autre part, en plus d'améliorer la précision et la complétude du mécanisme de détection, la variabilité de la taille des séquences d'appel système permet de réduire le nombre de séquences nécessaires pour caractériser le comportement normal du processus.

Une autre proposition pour améliorer l'approche de Forrest et al. est de tenir compte de l'information temporelle [JL01]. En effet, ces travaux partent du constat que les écarts de temps entre chaque appel système sont relativement stables au sein des séquences d'appels système issues des mêmes portions de code. De ce fait, il est donc possible d'enrichir la signature comportementale d'un programme au niveau de ses appels système en tenant compte de cette information temporelle. Pour cela, durant la phase d'apprentissage, pour chaque instance observée d'une même séquence d'appel système, sa signature temporelle est sauvegardée dans la base des séquences d'appels système pour la séquence qui lui est associée. Puis, une fois la phase d'apprentissage terminée, pour chaque séquence d'appel système, un modèle statistique considérant chacun des appels de la séquence comme une variable aléatoire est utilisé pour en décrire l'ensemble de distributions. Au final, c'est une base contenant l'ensemble des séquences d'appels système avec pour chacune d'entre elles l'ensemble de distributions des appels qui la composent qui est utilisé durant la phase de détection. À noter qu'afin de limiter le taux de faux positifs, il est nécessaire d'exclure l'information temporelle lorsque sa variance est trop élevée.

1.2.1.3 Machine à états finis

Une machine à états finis est un graphe orienté et étiqueté dont les sommets représentent des états et les arêtes les transitions entre ces états. On les appelle aussi parfois automates à états finis. Il existe un type bien particulier de machine à états finis appelé automate accepteurs. Ces derniers sont utilisés pour déterminer

si l'entrée qui leur est fournie est correcte ou non. Il est donc possible de modéliser un flot d'exécution à l'aide d'un automate de ce type, celui-ci pouvant ensuite être utilisé durant une phase de détection pour déterminer si le flot d'exécution observé est correct ou non.

Ce type de modélisation a été utilisé au niveau des appels système des processus [SBDB01]. Chaque état de l'automate représente le couple (numéro de l'appel système, adresse de l'appel). La phase d'apprentissage se fait à l'aide de traces d'exécution obtenues en utilisant l'outil *strace* qui permet notamment de connaître pour chaque appel le compteur ordinal du processus. C'est ce dernier qui contient l'adresse à partir de laquelle l'appel système a été effectué. Durant la phase de détection, à chaque fois qu'un appel système est effectué, on récupère le compteur ordinal du processus afin de construire un nouvel état. Puis, on regarde s'il existe au sein de l'automate à états fini une transition possible depuis l'état courant vers ce nouvel état. Si ce n'est pas le cas une alerte est alors levée. Enfin, le nouvel état construit devient le dernier état connu. Si celui-ci n'est pas présent dans l'automate, alors il est remplacé par un nœud initial à partir duquel tous les états valides sont atteignables. Ainsi, le processus normal de vérification reprendra au prochain état valide.

1.2.1.4 Modèle statistique

Un modèle de Markov caché est un modèle statistique décrivant l'évolution d'une variable aléatoire selon un processus de paramètres inconnus où il n'est pas nécessaire de connaître l'ensemble des états passés de la variable pour prédire probabilistiquement son prochain état. Il est donc possible d'utiliser un modèle de Markov caché pour modéliser l'enchaînement des appels système exécutés. Pour réaliser la modélisation, on analyse un grand nombre de traces d'exécution des programmes durant une fenêtre de temps où l'on considère que ces derniers sont en situation de fonctionnement normal. Le modèle ainsi créé est ensuite utilisé à l'exécution durant la phase de détection afin de lever une alerte lorsque qu'un appel système observé n'a pas été prédit. Notons que puisqu'il s'agit d'un modèle statistique, un seuil doit donc être défini pour savoir à partir de quelle probabilité une alerte doit être levée. Ce paramètre influence donc le taux de faux positifs et de faux négatifs de ce type d'approche.

Les travaux en détection d'intrusion qui reposent sur les modèles de Markov cachés [GRS06] ont montré que ce type de modélisation permet d'obtenir un taux de détection comparable à celui obtenu par les approches précédentes mais avec une surcharge à l'exécution plus faible. Notons également que des travaux ont cherché à utiliser ce type de modélisation pour détecter spécifiquement des élévations illégales du niveau de privilèges [CP03]. Le niveau de privilèges fait effectivement partie des informations connues du système d'exploitation à propos d'un processus. Ce mécanisme de détection cible les programmes capables d'élever eux-même leur niveau de privilèges. Comme les autres mécanismes de détection en boîte noire, il ne

détecte donc que les intrusions qui modifient l'enchaînement des appels système exécutés. Toutefois, le taux de détection des intrusions est bien meilleur en comparaison lorsque celles-ci aboutissent à une élévation illégale du niveau de privilèges.

1.2.2 Les approches de type boîte grise

Les approches de type boîte grise consistent à compléter les informations accessibles au niveau de l'interface entre les processus et le système d'exploitation par l'adjonction d'informations issues du fonctionnement interne des processus mais néanmoins aisément accessible au niveau du système. C'est par exemple le cas du contexte d'exécution ou encore des arguments passés en paramètres aux appels système. Par rapport aux approches de type boîte noire, ce type d'information va rendre plus difficile l'utilisation de mécanisme de contournement mais aussi permettre de détecter des attaques qui ne modifient pas l'enchaînement des appels système exécutés. L'amélioration de la détection va bien sûr dépendre du type d'information utilisé pour compléter le modèle de référence.

En pratique, la plupart des approches en boîte grise cherchent à compléter le contrôle des séquences d'appels système présenté par Forrest et al. [FHSL96, HFS98]. En effet, cette approche s'est déjà vu proposer des améliorations mais celles-ci constituaient toujours en une approche de type boîte noire (voir section 1.2.1.2). C'est pourquoi, même améliorés, ces modèles de détection restent peu efficaces contre les attaques qui ne modifient pas l'enchaînement des appels système exécutés et peuvent toujours être attaqués via des mécanismes de contournement, notamment par des attaques de type mimétisme [KKMR05, PSJ07]. Nous allons maintenant présenter plusieurs méthodes de modélisation comportementale en boîte grise et leurs apports respectifs en terme de détection.

1.2.2.1 Contrôle des appels aux bibliothèques

Une première manière d'appliquer en boîte grise le modèle de détection par contrôle des séquences est de déplacer ce contrôle des appels systèmes aux appels aux bibliothèques de fonctions [LJ01]. En effet, les appels aux bibliothèques de fonctions peuvent permettre de modéliser plus justement le comportement du programme que les appels système. Il est par exemple possible de trouver deux séquences différentes d'appels aux bibliothèques de fonctions qui produisent la même séquence d'appels système (on ne tient compte ici que du numéro des appels système). De plus, les bibliothèques de fonctions, contrairement aux appels système ne dépendent pas du système d'exploitation et permettent donc de créer des modèles de comportement portables.

Le mécanisme de construction de la base des séquences est identique à celui proposé par Forrest et al. (voir section 1.2.1.2). De même pour le mécanisme de détection. En pratique, la différence réside dans l'utilisation de la table d'importation des processus et de la table d'exportation des bibliothèques de fonctions liées

dynamiquement pour intercepter les appels de fonctions. Bien sûr cette approche n'est pas valable pour les bibliothèques de fonctions liées statiquement. Bien que cette première amélioration du modèle de Forrest et al. ne permet pas de détecter d'autres types d'attaques que ceux qui modifient le flot d'exécution du programme, le taux détection pour ces derniers est amélioré. Cependant, les mécanismes de contournement utilisables au niveau des appels systèmes le restent au niveau des appels aux bibliothèques de fonctions.

1.2.2.2 Contrôle des paramètres

Une autre manière d'améliorer la détection par le contrôle des séquences d'appels système via l'apport d'informations issues du fonctionnement interne des processus est de compléter le modèle de séquences par les arguments qui leurs sont passés en paramètre [KMVV03]. Pour cela, un profil des paramètres est construit pour chaque appel système présent dans les séquences. La difficulté de construction de ces profils réside dans la gestion du non-déterminisme. En effet, certains paramètres des appels systèmes peuvent systématiquement varier d'une exécution à une autre. Il n'existe donc pas de jeu de données qui permet de modéliser l'ensemble de valeurs de ce type de paramètre.

Pour répondre à ce problème, deux mécanismes sont employés suivant qu'il s'agisse d'entiers ou de chaînes de caractères. D'abord, dans le cas d'entiers, on renseigne manuellement pour chaque appel système quels paramètres ne doivent pas être utilisés pour créer le profil (par exemple, le numéro de port local lors d'une connexion *TCP*). Puis, dans le cas des chaînes de caractères, on utilise un algorithme de découverte de motifs pour caractériser le paramètre en question (par exemple, celui-ci doit toujours commencer par */var/www/html*).

Des travaux proposent d'aller plus loin et de tenir compte de la temporalité dans l'analyse des arguments des appels système afin de mettre en évidence certains flux d'information [BCS06]. Par exemple, si l'appel système n est un appel à *open* et que le descripteur de fichier retourné possède l'indice x, alors le paramètre du prochain appel à *close* doit être égal à x. L'analyse proposée est également capable de détecter des liens de dépendances au niveau des chaînes de caractères et notamment des chemins de fichiers. Par exemple, si l'appel à *opendir* avant une boucle se fait avec la chaîne de caractères */var/www*, alors tous les appels à *open* dans cette boucle auront comme paramètre un accès en lecture à des fichiers dont le chemin d'accès commence par */var/www*.

D'une part, l'ajout dans le modèle des arguments passés en paramètre des appels système permet alors d'améliorer la détection lorsque l'enchaînement des appels système exécutés est modifié. Il est par exemple plus difficile d'avoir recours à un mécanisme de contournement tel qu'une attaque par mimétisme qui devra alors aussi déjouer le contrôle des paramètres. D'autre part, la détection est maintenant possible lorsque seulement le flux des informations est modifié. En effet, le contrôle

des arguments passés en paramètre des appels système rend possible la détection des attaques contre les données de calcul qui ne modifient par le flot d'exécution du programme.

1.2.2.3 Contrôle du contexte d'exécution

Une autre manière de compléter un modèle de détection basé sur le contrôle des séquences d'appels système est d'y ajouter comme information issue du fonctionnement interne du programme le contexte d'exécution des appels système effectués. Ces informations sont extraites de la pile au moment de l'appel système. C'est par exemple le cas de la pile d'appels internes qui a mené à un appel système en particulier. Pour cela, lors de la phase d'apprentissage ainsi que lors de la phase de détection, la pile du processus est analysée pour déterminer les adresses internes des précédents appels de fonction dans la pile d'appel ayant mené à l'appel système observé. À noter toutefois que certains appels de fonction n'apparaissent pas dans ce modèle : ceux qui ne conduisent à aucun appel système.

Une fois ces informations extraites, des travaux proposent d'améliorer l'approche présentée à la section précédente [MRVK07]. Pour cela, le profil des arguments passés en paramètre des appels système n'est plus construit uniquement pour chaque appel, mais pour chaque contexte d'exécution de chaque appel système. Avant amélioration, ce modèle permettait déjà de détecter des intrusions qui modifient le flot d'exécution et des intrusions qui modifient le flux d'information. L'ajout de ce type d'informations permet d'améliorer le taux de détection et de rendre beaucoup plus difficile l'utilisation de techniques de contournement de type mimétisme.

À noter que d'autres travaux proposent d'utiliser les informations de la pile d'appel pour créer un graphe de contrôle dont les états ne modélisent pas seulement les appels système mais également les appels de fonction internes dans le processus [GRS04]. Il s'agit d'une amélioration de l'approche en boîte noire reposant sur un automate à états finis (voir section 1.2.1.3). Là encore, le type d'attaque détectée par l'approche améliorée ne change pas, mais le taux de détection ainsi que la difficulté pour contourner le mécanisme augmentent.

1.2.3 Les approches de type boîte blanche

Les approches de type boîte blanche sont celles qui permettent d'utiliser pour la phase de modélisation du comportement du programme l'ensemble des informations utilisées en interne par les processus. C'est par exemple le cas des algorithmes utilisés ou encore des structures de données internes qu'ils manipulent. Cette information peut provenir de la spécification de l'application ou être extraite de manière automatique de son code source. Nous présentons dans la suite de cette section ces deux approches pour la modélisation.

1.2.3.1 Analyse de la spécification

Il est juste de considérer que la spécification d'un programme fait partie de sa politique de sécurité. En effet, une intrusion réussie au sein d'un programme constitue nécessairement une déviation par rapport à sa spécification. *BMSL* est un langage de spécification conçu pour être utiliser pour la détection d'intrusion [US01]. Il repose sur l'utilisation d'expressions régulières pour décrire des suites d'évènements. Plus précisément, cette approche se concentre sur les appels système. Pour un programme donné, grâce à la connaissance de la spécification de ce dernier vis-à-vis du système d'exploitation, ces travaux utilisent le langage *BMSL* pour décrire les suites possibles d'appels système pour ce programme. Cela permet par exemple de spécifier que le programme ne peut pas ouvrir puis fermer un fichier sans au moins avoir effectué une opération de lecture ou d'écriture sur ce dernier. Des restrictions supplémentaires peuvent porter sur les paramètres des appels système afin de coller au mieux à la spécification réelle du programme. Par exemple, si le premier paramètre de *open* est le fichier */etc/passwd*, alors le mode d'accès ne peut que être *O_RDONLY*.

Cette approche a été testée sur le jeu de données du *DARPA* [LHF+96]. Les résultats ont montré que celle-ci permet de détecter de nombreuses attaques sans connaissance *a priori* du mode opératoire. Une approche similaire a été proposée dans le cadre des systèmes distribués [KRL97]. À nouveau, cette approche se concentre sur les opérations de base d'un programme (lecture, écriture, modification des propriétés, création de processus et intervention sur les processus existants) en permettant de décrire toute les contraintes possibles liant ces opérations. Cependant, ces travaux ajoutent la possibilité de décrire des contraintes liant deux processus différents. Par exemple, un processus B ne peut ouvrir en lecture le fichier X qu'une fois que le processus A a ouvert ce même fichier en écriture, effectuer au moins une opération d'écriture dessus puis l'a refermé. Cette approche a été testée hors ligne sur les traces d'exécution de programmes se synchronisant entre eux. Celle-ci a permis de mettre en évidence des attaques qui n'auraient pas pu être détectées sans tenir compte de la problématique de la synchronisation.

Ces deux travaux on montré que les approches reposant sur la connaissance de la spécification permettent d'obtenir de très bon taux de détection, avec peu de faux positifs levés et une faible surcharge à l'exécution. Toutefois, la phase de rédaction de la spécification n'est que rarement faite par les développeurs eux-même. Cette phase doit alors être effectuée pour l'ensemble des programmes du système que l'on cherche à surveiller. Il s'agit d'une part d'une tâche fastidieuse pour l'administrateur du système. D'autre part, l'aspect manuel de cette étape implique que celle-ci est donc susceptible d'introduire des informations erronées dans la spécification.

1.2.3.2 Analyse du code source

Si l'on ne dispose pas de la spécification ou que celle-ci n'est pas exploitable de manière automatisée, on peut se tourner vers le code source de l'application

pour essayer d'obtenir une approximation de sa spécification. L'objectif est donc d'analyser le code source du programme pour en dériver une spécification plus ou moins précise. Bien sûr, si le programme est vulnérable, alors son code source contient des erreurs et celles-ci sont susceptibles de se retrouver dans cette spécification.

Une première analyse facile à réaliser et facile à contrôler à l'exécution repose sur le contrôle des appels système [WD01]. Cette approche consiste à créer un graphe des appels système présent dans le code source puis à vérifier durant l'exécution du programme que les séquences d'appels système correspondent bien à ce graphe de référence. La figure 1.2 présente à gauche un exemple simple de code *C* et à droite le graphe des appels système correspondant. L'analyse réalisée est simple, on ne tient pas compte des valeurs des variables pour réduire les chemins possibles dans le graphe (dans l'exemple précédent, *getuid* et *geteuid* sont nécessairement appelés dans cet ordre).

Cette approche ne permet de détecter que des intrusions qui modifient l'enchaînement des appels système exécutés. En cela elle est similaire aux approches en boîte noire et basées sur le modèle de Forrest et al. (voir section 1.2.1.2). Cependant, le code source donné à analyser peut ne pas contenir tous les appels système que le processus va être amené à exécuter. C'est par exemple le cas si des appels système sont effectués par des fonctions dont le code n'est pas fourni car celles-ci sont implémentées dans des bibliothèques de fonctions. Toutefois, si le code donné à analyser contient bien tous les appels système que peut exécuter le processus, alors cette approche aura l'avantage de ne jamais lever de faux positifs.

Des travaux proposent de reprendre ce principe en augmentant la précision de l'analyse aux appels aux bibliothèque de fonctions [GSV05]. En effet, en pratique, peu de codes source utilisent directement les appels système. La plupart reposent sur l'utilisation de bibliothèques de fonctions qui, elles, effectuent des appels système. Cependant, même à ce niveau, il est possible de recourir à des mécanismes de contournement pour mettre en échec le système de détection d'intrusion. Concrètement, l'attaquant conduit la même analyse que celle utilisée pour construire le modèle de détection et modifie son attaque en conséquence afin que cette dernière corresponde au modèle [KKMR05, PSJ07].

Pour répondre à ce problème, le contrôle de l'intégrité du flot d'exécution (ou *CFI*, *Control-Flow Integrity*), bien que reposant sur le même principe, propose de maintenant contrôler l'ensemble des branchements dans le programme [ABEL05]. Cette fois, le graphe de contrôle permet de vérifier par exemple qu'un appel de fonction ou un appel système est bien effectué à partir d'une adresse source valide et que suite au retour de l'appel, l'adresse de destination est aussi valide. Concrètement, il n'est plus possible pour le programme de dévier vers un chemin d'exécution qui n'existe pas d'après son code source sans qu'une alerte ne soit levée. Cependant, il est toujours possible d'exploiter une vulnérabilité dans un programme au travers d'un chemin d'exécution valide (voir section 2.1). Pour cela, l'attaquant ne va pas cibler les données qui permettent de modifier le flot d'exécution du programme, mais

```
01. int f(int x){
02.
03.   if(x == 0){
04.
05.     return getuid();
06.   }
07.   else{
08.
09.     return geteuid();
10.   }
11. }
12.
13. void g(){
14.
15.   int fd;
16.   int id;
17.
18.   fd = open("/tmp/file", O_RDONLY);
19.   id = f(0);
20.
21.   close(fd);
22.   id = f(1);
23.
24.   exit(0);
25. }
```

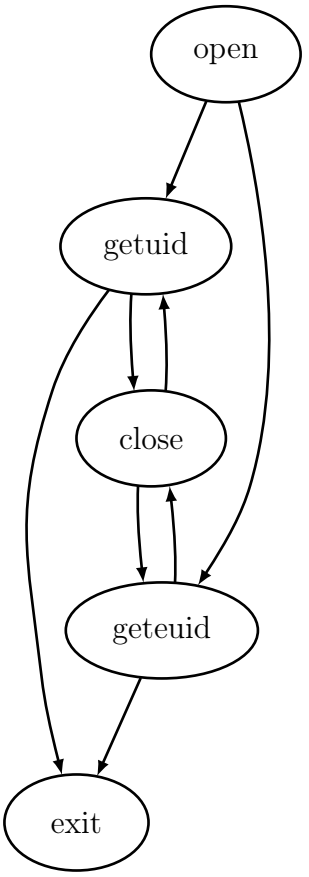


FIGURE 1.2 – Exemple de graphe des appels système

celles qui permettent d'en modifier son flux d'information. C'est par exemple le cas des données qui influencent la valeur des paramètres des appels de fonction sensibles tels que les appels système ou les appels aux bibliothèques de fonctions.

Pour détecter ce type de compromission durant l'exécution d'un programme, des travaux proposent d'effectuer un type particulier de suivi des flux d'information, le contrôle de pollution des données ou *tainted analysis*. Ce type d'approche va effectuer une forme de contrôle d'intégrité des données sensibles en vérifiant que les opérations utilisées pour générer la valeur de ces données sensibles ne reposent que sur des données sûres. Ce n'est par exemple pas le cas si des données brutes fournies par un utilisateur non-privilégié sont utilisées par ces opérations. S'il s'agit par contre de données fournies par le programme, par l'administrateur ou par un utilisateur non-privilégié lorsque ces dernières sont vérifiées par un mécanisme de contrôle, alors le résultat de l'opération est considéré comme intègre.

Dans sa forme la plus simple, le contrôle de pollution des données définit les niveaux de sécurité de manière binaire : seules les données provenant de sources considérées comme non-intègres sont marquées comme étant polluées, les autres don-

nées ne sont pas marquées car considérées comme intègres. Notons que des travaux ont exploré le *positive tainting* [HOM06]. Cette fois, seules les données considérées comme intègres sont marquées comme sûres, les autres données ne sont pas marquées car considérées comme polluées. Ces approches sont le plus souvent obtenues par instrumentation des programmes à surveiller. Celles-ci se prêtent donc bien aux applications *web*. Il est en effet facile dans ce cas d'identifier les information non-intègres (les données envoyées par le client *web*) et les information sensibles (les données retournées au client *web* mais par exemple aussi l'exécution de requêtes *SQL*). Des implémentations ont ainsi été proposées pour *Perl* [Sch00], pour *PHP* [NTGG+05], pour *Ruby* [TH05] ou encore pour *Java* [HCF05]. Notons que des travaux ont cherché à généraliser ce type d'approche au niveau des systèmes d'exploitation [Mad00].

La principale limitation de ce type d'approche réside dans le fait que dans le cas général, la procédure d'instrumentation ne peut se faire sans intervention humaine. En effet, dans le cadre du fonctionnement normal d'une application, même de type *web*, non seulement il n'est pas toujours possible d'identifier de manière automatique les opérations sensibles, mais de plus un certain nombre de ces opérations dépend de données fournies par des utilisateurs non-privilégiés. Comme il n'est pas envisageable de lever une alerte à chacune de ces opérations, la procédure d'instrumentation ne peut être entièrement automatisée. Ces données non-intègres devront être contrôlées par l'application et l'administrateur précisera au système de détection d'intrusion que celles-ci ne doivent pas être considérées comme polluées.

Une manière de contrôler les flux d'information de manière automatique consiste à se concentrer sur les opérations plutôt que sur les données. En effet, par exemple lors d'une attaque exploitant un dépassement de tampon, certes il y a un flux d'information invalide entre deux données qui normalement ne dépendent pas l'une de l'autre. Mais, bien souvent, l'opération à l'origine de ce flux d'information invalide n'est pas censée modifier la donnée ciblée d'après la spécification du programme. Pour détecter cela, une méthode analogue au contrôle de l'intégrité du flot d'exécution abordée précédemment a été proposée. Il s'agit du contrôle de l'intégrité du flux d'information (ou *DFI*, *Data-Flow Integrity*). Cette approche repose sur la création d'un graphe des flux d'informations autorisés durant l'exécution du programme [CCH06]. Concrètement, à un instant donné, lorsqu'une donnée est lue, d'après le graphe, seul un petit nombre d'instructions peut avoir été responsable de l'écriture de cette donnée. En vérifiant la validité de l'instruction qui a écrit en dernier la donnée en cours de lecture, il n'est plus possible, sans lever une alerte, de détourner une instruction valide pour lui faire corrompre une donnée qu'elle n'est pas censée modifier.

1.2.4 Positionnement des travaux

L'approche que nous proposons dans ces travaux utilise un modèle de comportement normal pour détecter les intrusions qui ciblent les applications. Cependant,

nous avons pour objectif de nous concentrer sur les attaques qui ne forcent pas le flot d'exécution du programme vers un chemin invalide (voir section 2.1). En effet, il existe déjà des approches efficaces et avec un bon rapport entre le taux de détection et la surcharge à l'exécution pour les autres attaques, celles qui forcent le flot d'exécution du programme à dévier de tout chemin valide.

Le contrôle de l'intégrité du flux d'information [CCH06] est une approche efficace contre le type d'attaques qui nous intéresse. À l'instar de cette dernière, l'approche que nous proposons adopte un point de vue de type boîte blanche et repose sur l'analyse statique du code source pour construire son modèle de détection. Toutefois, il reste possible de contourner le mécanisme de détection *DFI* en utilisant de manière illégale une des instructions autorisées dans la liste des instructions associées à une opération de lecture.

L'approche que nous proposons ne se concentre pas sur les flux d'information mais sur la valeur des données utilisées par le processus. Ceci permet de cibler les attaques que ne pourrait détecter le mécanisme de détection *DFI*. De plus, bien que ce dernier soit capable de cibler aussi bien les attaques qui ciblent les données de calcul que celles qui ciblent les données de contrôle, celui-ci induit une surcharge importante à l'exécution. Nous avons choisi de ne cibler que les attaques contre les données de calcul avec pour objectif d'obtenir une surcharge faible à l'exécution afin de pouvoir compléter d'autres approches qui elles aussi induisent une surcharge faible à l'exécution mais qui ciblent principalement les attaques contre les données de contrôle.

1.3 La détection d'erreur

L'approche que nous proposons dans ces travaux de thèse consiste en la détection des états erronés des applications lorsque ceux-ci sont provoqués par des attaques. Or, la détection d'erreurs est un problème déjà traité par de nombreux travaux antérieurs dans le domaine de la sûreté de fonctionnement. Dans cette section, nous présentons d'abord ce qui susceptible de constituer des entraves à la sûreté de fonctionnement et qui correspond donc à des concepts fondamentaux de ce domaine, à savoir les notions de faute, d'erreur et de défaillance. Des travaux issus du domaine de la sûreté de fonctionnement ont également proposé des techniques de détection d'erreur. Le mécanisme de détection des intrusions que nous proposons dans cette thèse s'inspire de ces travaux. C'est pourquoi dans la suite de cette section nous présentons la technique issue de ce domaine qui a inspiré notre mécanisme de détection.

1.3.1 Définitions de la sûreté de fonctionnement

La sûreté de fonctionnement d'un système informatique est la propriété qui permet à ses utilisateurs de placer une confiance justifiée dans le service qu'il leur

délivre [ABC+95]. Le service délivré par le système informatique est son comportement tel que perçu par ses utilisateurs. Notons que l'on appelle utilisateur d'un système informatique tout autre système qui interagit directement avec lui. Celui-ci n'est pas nécessairement humain, il peut s'agir par exemple d'un autre système informatique. La sûreté de fonctionnement d'un système informatique peut être caractérisée par un certain nombre d'attributs.

On trouve d'abord les notions de confidentialité, d'intégrité et de disponibilité précédemment utilisées pour décrire une politique de sécurité dans la section 1.1. La notion de sécurité fait donc partie intégrante de la sûreté de fonctionnement. À cela on ajoute la notion de fiabilité, qui est l'aptitude à respecter la continuité de service, et la notion de maintenabilité, qui est l'aptitude aux réparations et aux évolutions. Ces attributs permettent d'exprimer les propriétés qui sont attendues du système et d'apprécier les moyens mis en œuvre pour assurer leur respect.

Ces attributs peuvent être mis en défaut par des entraves au sein du système informatique considéré. Bien qu'indésirables, ces entraves ne sont pas inattendues et sont la cause ou le résultat de la perte de la confiance placée par les utilisateurs du système informatique dans le service délivré. Ces entraves à la sûreté de fonctionnement peuvent être décrites à l'aide de trois concepts nécessaires et suffisants, à savoir les notions de faute, d'erreur et de défaillance. Nous abordons plus en détails ces trois concepts un peu plus loin dans la suite de cette section.

Ces entraves forment au sein d'un composant du système considéré une chaîne causale : une défaillance est causée par un état erroné du système qui lui-même a pour origine l'activation d'une faute. De plus, cette chaîne causale est susceptible de se propager à d'autres composants. Par exemple, la défaillance d'un composant peut activer une faute dans un autre composant qui dépend de lui. Cette chaîne causale peut également être étendue aux menaces vis-à-vis de la sécurité des systèmes d'information. On parle alors du trio : vulnérabilité - attaque - intrusion.

En effet, une intrusion dans un système d'information implique nécessairement que celui-ci est défaillant. Cette intrusion a été causée par une attaque contre le système qui a ainsi placé ce dernier dans un état erroné. Une vulnérabilité présente dans le système est ce qui a rendu possible cette attaque. De même, une intrusion dans un composant d'un système d'information peut permettre d'exploiter une vulnérabilité présente dans un autre composant et qu'il n'est pas possible d'exploiter directement. Par exemple, une intrusion dans un service distant s'exécutant avec peu de privilèges va permettre à l'utilisateur malveillant d'exploiter une vulnérabilité présente dans un service local s'exécutant avec des droits administrateur.

1.3.1.1 Faute

Dans ces travaux de thèse, nous ne considérons que les fautes des composants logiciels des systèmes d'information. Une faute logicielle peut avoir une origine matérielle (par exemple, la défaillance d'un composant matériel sur lequel repose le composant

logiciel concerné) ou bien une origine humaine (par exemple, une défaillance du processus de configuration du composant logiciel concerné). Dans le premier cas, la faute est introduite par l'environnement du composant logiciel pendant sa phase de fonctionnement. Dans le second cas, la faute est présente dès le début de la phase de fonctionnement car introduite en amont.

Une faute correspond à ce qui est susceptible de créer une erreur au sein du système considéré. Mais ceci n'est pas systématique. C'est par exemple le cas d'une faute présente dès le début de la phase de fonctionnement mais qui n'a pas encore eu l'occasion de mettre le système d'information dans un état erroné. On dit alors que la faute est dans un état dormant. Une faute donnée possède donc deux états possibles : dormant ou actif. Une faute est dite active lorsque celle-ci a effectivement engendré une erreur au sein du système considéré. Cette situation ne se produit que si la faute est introduite par l'environnement du composant logiciel durant son exécution ou bien si une faute dormante préalablement présente dans le composant est activée durant l'exécution.

L'activation d'une faute dormante peut être déclenchée par un utilisateur du système. Cette activation peut être accidentelle ou bien intentionnelle. Dans ces travaux de thèse portant sur la détection d'intrusion, nous nous intéressons donc uniquement aux activations intentionnelles. Prenons comme exemple le cas d'une vulnérabilité présente dans un programme particulier. Cette vulnérabilité provient d'une défaillance du processus de développement du programme considéré. Dans cet exemple, la faute est dans un état dormant jusqu'à ce qu'un utilisateur malveillant exploite la vulnérabilité. L'attaque contre le programme correspond bien a une activation intentionnelle.

1.3.1.2 Erreur

Une erreur est donc provoquée par l'activation d'une faute et celle-ci correspond à un état du système susceptible d'entraîner une défaillance. Là encore, cela n'est pas quelque chose de systématique. Tant que le système considéré est dans un état erroné mais que toutefois aucune défaillance n'a encore été provoquée, on parle alors d'erreur latente. Toutefois, si le système possède des mécanismes de détection d'erreur, une erreur ne sera plus considérée comme latente si celle-ci est détectée bien qu'elle n'ait pas encore provoqué de défaillance. La capacité d'une erreur à engendrer une défaillance dépend bien sûr de la définition des propriétés attendues du système à l'aide des attributs présentés précédemment. Mais elle dépend aussi de la composition du système lui-même ainsi que de ses activités.

En effet, ces deux paramètres peuvent influencer la capacité d'une erreur latente à provoquer finalement une défaillance. Première situation, un système d'information donné peut dans sa composition avoir recours à des mécanismes de détection et de tolérance aux erreurs. Dans ce cas, il est possible qu'une erreur latente soit détectée puis corrigée avant de pouvoir engendrer une défaillance. Deuxième situation, les

activités nécessaires au bon fonctionnement des services rendus par le système peuvent aussi avoir pour effet de neutraliser involontairement une erreur latente avant que celle-ci n'ait pu engendrer une défaillance.

Prenons par exemple le cas d'un système d'information qui réalise régulièrement des tâches courtes à l'aide de processus légers. Supposons que l'un des processus ait été mis dans un état erroné latent. Si la tâche à réaliser est finie avant que l'erreur n'ait pu engendrer une défaillance, alors l'activité normale du système d'information va remettre ce dernier dans un état correct en terminant le processus léger correspondant. Bien sûr, il est possible qu'une erreur latente se propage avant d'être corrigée et puisse ainsi créer de nouvelles erreurs latentes. Le système d'information ne peut alors se trouver à nouveau dans un état correct que si les éventuels mécanismes de détection et de tolérance aux erreurs ou son activité normale arrivent à remettre le système dans l'état attendu d'après sa spécification.

1.3.1.3 Défaillance

Une défaillance du système est donc provoquée par une erreur et correspond au moment où le service délivré par le système dévie du service attendu. Autrement dit, on considère un système informatique défaillant lorsqu'un ou plusieurs des attributs utilisés pour exprimer les propriétés qui sont attendues du système ne sont plus respectées. Une défaillance est donc une erreur observable par un utilisateur du système. La défaillance est l'aboutissement de la chaîne causale abordée précédemment : la défaillance est le résultat d'un état erroné du système lui-même provoqué par l'activation d'une faute. Une défaillance peut se propager : la défaillance d'un composant du système est une faute pour le système considéré ou pour tout autre composant qui dépend de lui.

On peut différencier les défaillances d'un système selon plusieurs caractéristiques. Le choix de ces caractéristiques va dépendre de la perception de la criticité pour le système considéré. Par exemple, on peut choisir de considérer qu'un système peut être dans un état de défaillance transitoire ou durable. Dans le cas d'un système durablement défaillant, les services rendus seront erronés de manière persistante. Autre exemple, on peut choisir de considérer qu'un système peut être dans un état de défaillance cohérent ou incohérent. Dans le cas d'un système qui est dans un état de défaillance cohérent, la perception des erreurs par les utilisateurs du système est identique pour tous.

Les défaillances peuvent être évitées grâce à des mécanismes de prévention ou de tolérance aux fautes. Dans ce cas, un mécanisme de détection des erreurs est bien souvent préalablement nécessaire. On peut alors envisager d'utiliser un mécanisme de ce type pour chercher à détecter les erreurs à l'origine des intrusions. En effet, dans le contexte qui nous intéresse, à savoir la détection d'intrusion, la chaîne causale se manifeste de la manière suivante : d'abord, une faute logicielle dormante dans un programme, la vulnérabilité, va être activée par un utilisateur malveillant exploitant

cette vulnérabilité au travers d'une attaque. Puis, l'état erroné du processus en cours d'exécution va permettre une intrusion au sein du système sur lequel celui-ci s'exécute, le plaçant ainsi dans un état de défaillance intentionnel.

Dans le cadre des systèmes distribués, le projet *MAFTIA* [AAC+03, VNC03, WWRS03, SWWR04] présente une approche pour la tolérance aux fautes qui est adaptée aux contraintes de la sécurité. En effet, le modèle de fautes ainsi proposé permet de prendre en compte les fautes accidentelles mais aussi les fautes intentionnelles. Dans ces travaux, nous proposons de détecter les erreurs à l'origine des intrusions. Plus précisément, pour détecter une intrusion, ou une tentative d'intrusion, nous proposons de détecter l'état erroné du programme suite à l'exploitation de la vulnérabilité par un utilisateur malveillant. À l'instar des travaux du projet *MAFTIA*, nous allons donc nous reposer sur un mécanisme de détection d'erreurs. Nous présentons donc dans la section suivante un mécanisme de détection d'erreur issu du domaine de la sûreté de fonctionnement et qui a inspiré le mécanisme de détection que nous proposons ici.

1.3.2 Contrôles de vraisemblance

Les contrôles de vraisemblance consistent en l'évaluation des objets manipulés en interne par le programme. Ces évaluations sont effectuées pendant l'exécution du programme et ont pour objectif de vérifier une expression logique. Si l'expression logique n'est pas vérifiée, alors le programme est considéré comme étant dans un état erroné et une alerte est émise.

Le principal avantage des contrôles de vraisemblance est de ne générer qu'un faible sur-coût à l'exécution. Ceci est dû au fait que le nombre d'instructions nécessaires pour vérifier l'ensemble des expressions logiques d'un composant est le plus souvent faible par rapport au nombre total d'instructions de ce même composant. Ils peuvent être utilisés pour détecter un grand nombre de types de fautes différents mais leur couverture est généralement faible.

Les contrôles de vraisemblance peuvent être implémentés à différents niveaux, soit par des composants matériels, soit par des composants logiciels. Dans les composants matériels, ces contrôles permettent de détecter des erreurs dans les données utilisées à bas niveau, tel que le code d'une instruction ou une adresse mémoire, ou des violations de la segmentation des espaces mémoires mis en œuvre par des composants tels qu'une unité de gestion de mémoire. Dans les composants logiciels, ces contrôles permettent de détecter des erreurs dans les données utilisées à plus haut niveau, telles que les données de calcul, par exemple en vérifiant la conformité de ces dernières vis-à-vis d'invariants.

1.3.2.1 Utilisation d'un moniteur externe

Au niveau matériel, les contrôles de vraisemblance sont effectués par un composant externe appelé moniteur. Des travaux ont cherché à adapter cette technique

pour l'utiliser également au niveau logiciel. C'est par exemple le cas des violations de la segmentation de l'espace mémoire qui peuvent être détectées sans aucune modification du processus [KKA95].

Pour s'assurer que le code exécuté est correct, des moniteurs capables de contrôler le flot d'exécution des processus ont été proposés [MM88, SM90, BCNP03]. Dans ce cas le programme est modifié pour embarquer des sommes de contrôle. Ces sommes de contrôle sont des données passives et ne modifient pas le comportement du programme. Chaque bloc de code (c'est-à-dire chaque groupe d'instructions sans branchement) possède une somme de contrôle. Celle-ci est contrôlée par le moniteur avant d'exécuter le bloc de code correspondant. Une somme de contrôle permet de vérifier que les blocs de code s'enchaînent dans le bon ordre. Le moniteur contrôle ainsi que le flot d'exécution se déroule sur un chemin valide.

1.3.2.2 Programmation défensive

La programmation défensive est une approche de développement où l'on suppose que le code peut contenir des fautes non détectées ou des incohérences. Ces fautes, initialement introduites dans le code de manière non-intentionnelle, peuvent être détectées par l'ajout de code supplémentaire. Ce dernier a pour but de vérifier l'état du système à des moments précis afin d'en vérifier la cohérence. Lorsqu'une incohérence est détectée, ces mécanismes peuvent essayer de remettre le programme dans un état cohérent ou bien simplement lever une alerte. Dans le premier cas, le code ajouté est un mécanisme de tolérance aux fautes.

La programmation défensive est une manière d'implémenter au niveau des composants logiciels les contrôles de vraisemblance (voir section 1.3.2). Bien sûr, ces vérifications sont redondantes du point de vue de la spécification, mais elles permettent de détecter lorsque le programme a dévié de sa spécification, par exemple après qu'une défaillance ait eu lieu. Ces contrôles peuvent être utilisés pour vérifier l'exactitude des instructions en cours d'exécution par le programme ou pour vérifier la valeur des données manipulées par ces instructions. De tels contrôles peuvent alors être utilisés pour alerter l'administrateur d'un système d'information lorsque des processus voient leur flot d'exécution détourné ou leurs données corrompues.

Lorsque ces contrôles sont ajoutés au niveau du code source, on les appelle des assertions exécutables. Ces assertions sont déduites de la spécification du programme et permettent durant son exécution de s'assurer que son fonctionnement est normal. À noter qu'elles peuvent également être utilisées par un vérificateur de code pour s'assurer *a priori* que les conditions spécifiées par les assertions sont satisfaites pour toute exécution du système [Cap75, RCK05]. Des mécanismes à base d'assertions exécutables ont été proposés pour contrôler le flot d'exécution des programmes [MN95, ROSM02, GRRV03, VHM03]. Ces approches sont similaires aux mécanismes de contrôle du flot d'exécution reposant sur l'utilisation d'un moniteur externe. Cependant, cette fois le programme n'embarque pas uniquement les sommes

de contrôle mais également les instructions de vérification. Ce type d'approche requiert donc une phase d'instrumentation des programmes à surveiller. De plus, la vérification étant située dans les blocs de code à exécuter, ce type de mécanisme ne peut détecter une déviation du flot d'exécution qu'à destination d'un bloc valide.

En effet, un code injecté par un utilisateur malveillant ne contient bien sûr pas d'instructions de vérification et une déviation du flot d'exécution à destination de ce dernier n'est donc pas détecté par ce type de mécanisme. L'utilisation d'assertions exécutables peut être étendue à la vérification du flot de données [YWZK09]. Toutefois, dans ce cas les propriétés à vérifier ne sont pas déduites automatiquement du code source mais doivent être spécifiées par le développeur de l'application considérée.

1.3.3 Positionnement des travaux

Une intrusion dans un programme provoque une déviation comportementale de ce dernier. Il s'agit donc du point de vue de la sûreté de fonctionnement d'une défaillance du programme. L'état erroné du processus à l'origine de cette intrusion a été provoqué par une attaque qui peut donc à son tour être vue du point de la sûreté de fonctionnement comme l'activation d'une faute au sein du programme. Dans cette thèse, afin de détecter une intrusion, nous ne cherchons pas à la détecter une fois que cette dernière est effective (c'est-à-dire une fois que le programme présente une défaillance). Ce que nous proposons c'est de détecter les états erronés du processus susceptibles de conduire à une intrusion. Contrairement aux travaux du projet *MAFTIA* [AAC+03, VNC03, WWRS03, SWWR04], nous ne faisons pas de tolérance. Nous nous limitons à la phase de détection suivie de la levée d'une alerte.

Pour cela, nous déduisons par analyse statique du code source du programme des propriétés sur les données manipulées par le processus que pourrait violer une tentative d'intrusion. En effet, lorsqu'une telle violation est détectée, c'est que le processus est nécessairement dans un état erroné. À l'aide de contrôles de vraisemblance pour vérifier ces propriétés, il est alors possible de détecter une attaque au sein du programme durant son exécution. Puisque que nous disposons du code source du programme, ces derniers sont implémentés, à l'instar des techniques de programmation défensive, à l'aide d'assertions exécutables ajoutées au code source.

1.4 L'injection de fautes logicielle

L'injection de fautes logicielle est utilisée dans le domaine de la sécurité pour découvrir de nouvelles vulnérabilités et dans le domaine de la sûreté de fonctionnement pour évaluer et valider les mécanismes de détection et de tolérance aux fautes. Contrairement aux fautes matérielles qui peuvent être simulées logiciellement mais aussi matériellement (par exemple en allant perturber les signaux électriques directement au niveau des circuits et des composants), les fautes logicielles, elles, ne peuvent être

simulées que logiciellement. Dans tous les cas, la pertinence de l'approche réside dans la qualité de la modélisation du type de faute que l'on cherche à simuler.

Au niveau logiciel, une approche utilisée pour injecter de manière automatique des fautes au niveau des entrées d'un programme se nomme *fuzzing*. Dérivé du terme anglais *fuzzy* qui signifie confus, cette méthode consiste à fournir au programme des données confuses (c'est-à-dire en partie erronées). Il s'agit d'une méthode de test logiciel qui permet d'automatiser la découverte de fautes de conception dans des applications [SGA07] en essayant de trouver des jeux de données erronés que pourrait rencontrer un utilisateur du programme. Cela permet ainsi de tester la fiabilité du programme au niveau de ses points d'entrée et donc de tester les fonctions qui réceptionnent les données. En effet, bien qu'un programme puisse légitimement s'attendre à ce que les données qu'on lui envoie soient correctement formatées, ceci n'est aucunement garanti et la source émettrice ne doit pas être considérée comme de confiance. Cette approche peut s'appliquer à tous les points d'entrée du programme. Les plus ciblés sont en général ceux utilisés pour la lecture de fichiers formatés [FFUZZ] et le traitement des données reçues d'une autre application [COMR]. Toutefois, d'autres cibles telles que les entrées utilisateur [MFS90] ou les variables d'environnement [SFUZZ] peuvent être également testées par ce procédé. Cette approche facilite ainsi notamment l'identification de nouvelles vulnérabilités.

Cependant, cette approche pour l'injection de fautes présente des limites, notamment dans le cadre des applications réseau (par exemple, le chiffrement des communications rend difficile la génération de jeux de données erronés par mutation de jeux de données réels). C'est pourquoi, d'autres mécanismes plus sophistiqués, allant jusqu'à l'injection de fautes directement en mémoire ont été proposés. D'autres mécanismes choisissent de se restreindre à une classe de programme en incorporant certaines connaissances communes à un ensemble d'applications (par exemple, la connaissance d'un protocole réseau particulier). Dans tous les cas, les erreurs qui peuvent être découvertes par ces procédés entièrement automatiques doivent pouvoir être déclenchées par la corruption de certaines données et doivent nécessairement provoquer une défaillance observable de l'application.

Le mécanisme d'évaluation que nous proposons pour tester, de manière automatique, notre mécanisme de détection d'intrusion, repose sur un mécanisme d'injection en mémoire de fautes logicielles pour simuler le résultat d'une attaque contre une application donnée. Afin de situer l'approche pour l'injection de fautes que nous proposons, nous présentons donc dans la suite de cette section la procédure d'injection utilisée par différents outils d'injection de fautes ainsi que les différents modèles de fautes logicielles proposées jusqu'alors.

1.4.1 Procédure d'injection

Une procédure d'injection entièrement automatique, bien que moins précise, permet d'identifier des erreurs de conception plus rapidement que les approches tradi-

tionnelles qui consistent à auditer manuellement le code source ou à écrire des tests unitaires spécifiques. Bien sûr, le caractère automatique de la procédure implique que moins d'erreurs de conception seront détectées. Il s'agit donc en général d'une première étape avant de passer aux approches plus coûteuses en temps. Ce manque de précision est la conséquence du fait que l'utilisateur d'un outil d'injection de fautes entièrement automatique n'a besoin d'avoir aucune connaissance sur le fonctionnement interne du programme qu'il souhaite tester.

Tout au plus, seule la connaissance de ses interfaces peut éventuellement être nécessaire (voir section 1.4.2). Le programme à tester est alors vu comme une boîte noire par le mécanisme d'injection de fautes. Il est alors possible d'appliquer cette méthode sur des applications dont on ne dispose pas du code source. La recherche de vulnérabilité grâce à ce type d'approches, notamment par des mécanismes de *fuzzing*, est par conséquent de plus en plus utilisée par les développeurs et les chercheurs en sécurité mais aussi par les utilisateurs malveillants.

La procédure d'injection comprend deux phases importantes : la phase d'injection, c'est-à-dire le moment où la corruption des données utilisées par le programme en cours de test a lieu, et la phase de surveillance, c'est-à-dire tout le temps suite à l'injection où le mécanisme va collecter les informations nécessaires à l'analyse du résultat de l'injection. Nous allons maintenant détailler ces deux phases.

1.4.1.1 Phase d'injection

On peut distinguer deux catégories d'outil d'injection de fautes. Tout d'abord, il y a les outils conçus pour tester un programme en particulier. Ils peuvent par exemple être utilisés pour valider la phase de développement d'une application. C'est notamment le cas des outils de *fuzzing* qui peuvent être utilisés avant la phase de tests unitaires pour un logiciel particulier. Ces derniers sont en général capables de cibler un grand nombre de points d'entrée du programme en cours de test (et des points d'entrée de types différents).

Enfin, il y a les outils conçus pour tester toute une catégorie de programmes. Ils peuvent par exemple être utilisés pour évaluer la sécurité d'un ensemble d'applications implémentant toutes un même protocole réseau. Ce type de mécanisme d'injection de fautes se concentre le plus souvent sur un nombre restreint de points d'entrée. C'est par exemple le cas des outils de *fuzzing* utilisés pour tester les applications *web*.

Dans tous les cas, le mécanisme d'injection de fautes va répéter la phase d'injection un grand nombre de fois et ce jusqu'à obtenir le taux de couverture souhaité. Celui-ci peut dépendre en partie du modèle utilisé pour générer les données à injecter (voir section 1.4.2). Toutefois, on trouve notamment parmi les critères d'arrêt les plus simples la couverture du code ou la couverture du temps d'exécution. Il est toutefois possible de chercher à couvrir des critères plus complexes comme les chemins d'exécution possibles.

1.4.1.2 Phase de surveillance

Pour chaque phase d'injection, il y a en parallèle une phase d'observation. En effet, la surveillance du comportement du programme suite à l'envoi de données erronées est nécessaire pour pouvoir déterminer si l'injection a permis de mettre en évidence une faute de conception. Cette surveillance se réalise en deux étapes. Dans un premier temps, un certain nombre d'informations sur l'exécution du programme sont collectées. Puis, dans un second temps, ces informations sont analysées afin de détecter une éventuelle défaillance du programme. L'analyse peut également essayer de fournir des informations sur l'erreur à l'origine de la défaillance. Par exemple, l'analyse peut utiliser une trace d'exécution du programme sans injection pour déterminer si les traces observées durant la phase de tests représentent ou non un état de défaillance.

La première information qu'il est important de sauvegarder est bien sûr le jeu de données utilisé pour réaliser l'injection. En effet, afin de pouvoir localiser et corriger une erreur de conception mise en évidence par une injection, il est important de pouvoir reproduire la défaillance engendrée par une injection. À noter que pour faciliter la localisation de l'erreur de conception, il est peut être intéressant de chercher à reproduire la défaillance à l'aide d'un jeu de données minimal. Des outils de réduction des jeux de données permettent de faire cela de manière automatique à partir d'un jeu de données initial [DELTA, LITH].

La seconde information qu'il est important de sauvegarder est l'état du programme suite à l'injection. C'est ce type d'information qui permet de déterminer si une défaillance a eu lieu durant l'exécution du processus. En effet, si par exemple le programme s'arrête de manière inattendue ou bien au contraire se bloque, cela met en évidence le fait que l'injection a effectivement engendré une erreur à l'exécution. Notons qu'afin de détecter le blocage d'un programme, le mécanisme d'injection de fautes doit pouvoir déterminer que celui-ci ne répond plus. Notons également qu'il est possible d'avoir recours à un débogueur pour la collecte d'information. Cela permet de détecter des erreurs qui ne mettent pas en péril l'exécution du processus, telles que des fuites de mémoire.

1.4.2 Génération des données

Pour effectuer la phase d'injection, l'outil d'injection de fautes doit être capable de générer des données erronées à fournir au mécanisme d'injection. La qualité d'une campagne d'injection dépend d'ailleurs essentiellement de la pertinence et de la diversité des jeux de données générés. Différents modèles ont été proposés pour générer ces jeux de données erronées. Suivant l'approche sur laquelle ils reposent, ils nécessitent plus ou moins d'interventions de la part de l'utilisateur de l'outil d'injection de fautes. Les modèles qui requièrent le moins d'information de la part de l'utilisateur sont également ceux qui produisent le moins de résultats. Autrement dit, la qualité d'une campagne d'injection se fait au dépend de son automatisation.

Nous présentons dans la suite de cette section les différents modèles qui ont été proposés pour générer ces jeux de données erronées.

1.4.2.1 Modèle aveugle

Ce type de modélisation n'a aucune connaissance des formats de données attendus. L'approche la plus simple pour la réaliser consiste à envoyer aux entrées ciblées du programme en cours de test des données complètement aléatoires. Si bien sûr cette approche ne requiert aucune intervention de l'utilisateur, celle-ci ne permet qu'un champ d'actions limité. En effet, cette approche peut suffire pour des points d'entrée simples, par exemple gérant des chaînes de caractères, telles que l'entrée standard ou les arguments d'un programme en ligne de commande [MFS90]. Cependant, celle-ci n'aurait que peu d'utilité sur des points d'entrée plus complexes telle qu'une connexion réseau qui s'attend à réceptionner des données selon un protocole spécifique.

Pour pallier cet inconvénient, une approche plus élaborée consiste à rejouer des jeux de données valides après les avoir mutés. Pour réaliser cela, cette approche se déroule en deux temps. D'abord, une phase d'observation durant laquelle les données arrivant aux points d'entrée ciblés sont interceptées puis enregistrées avant d'être finalement transmises à l'application en cours de test. Puis, à partir de ces données interceptées, l'outil d'injection de fautes va générer par mutation de nouveaux jeux de données. L'opération de mutation peut être complètement aléatoire [PXFZ] ou bien réalisée à l'aide d'heuristiques [TAOF, GPF].

Toutefois, de tels mécanismes sont principalement limités aux interfaces de communication. En effet, sans aucune connaissance *a priori* du fonctionnement interne du programme, seules ces interfaces permettent d'enregistrer suffisamment d'informations pour ensuite générer par mutation des jeux de données erronées. De plus, même dans ce cas, cette approche reste encore limitée face à de nombreux protocoles de communication. Par exemple, ceux qui incluent des sommes de contrôle dans leurs communications ou bien qui nécessitent de répondre à un challenge pour l'authentification.

Notons que les modèles aveugles sont également utilisés pour simuler logiciellement des fautes matérielles. C'est par exemple le cas d'une faute de type corruption mémoire. En effet, il suffit de disposer d'un mécanisme ayant un accès direct à la mémoire du système informatique pour pouvoir reproduire le résultat d'une corruption mémoire. L'avantage de simuler ce type de faute au niveau logiciel est que cela permet de cibler la plage d'adresses d'un composant précis (par exemple, un programme) afin de tester ce dernier en particulier.

Le système d'injection *FERRARI* proposé par Kanawati et al. [KKA95] est un exemple de mécanisme permettant de reproduire ce type de faute pour un programme donné. Celui-ci permet également d'évaluer l'impact de l'injection sur le programme. Pour cela il commence par analyser le fichier exécutable afin de déter-

miner l'emplacement des sections de code et de données. Il effectue ensuite une première exécution du processus sans réaliser d'injection afin d'enregistrer une trace d'exécution du programme en situation de fonctionnement normal. Enfin, il procède à une série d'injections et observe le comportement du programme. La mesure de la couverture des injections est préalablement choisie. Elle est soit spatiale (couverture de l'espace mémoire) soit temporelle (couverture de la durée d'exécution). Pour effectuer une injection, le mécanisme choisit aléatoirement le moment de l'injection ainsi que l'emplacement mémoire. Le type de corruption est également préalablement choisi. Cinq modèles de fautes sont proposés : inverser un bit, forcer un bit à 1, forcer un bit à 0, forcer un octet à 1 et forcer un octet à 0.

1.4.2.2 Modèle spécifique

Dans de nombreux protocoles de communication, un paquet de données peut posséder des champs dont la valeur dépend d'un ou plusieurs autres champs de ce paquet. Par exemple une longueur de chaîne ou bien encore une somme de contrôle. De même, un paquet de données peut posséder des champs dont la valeur dépend d'un ou plusieurs autres paquets déjà échangés. Par exemple un compteur de paquets ou bien encore une réponse à un challenge d'authentification. C'est pour ces raisons que des mécanismes d'injection de fautes fondés sur la spécification attendue par le programme à tester ont été proposés.

Ce type d'approche va donc utiliser cette spécification pour générer les jeux de données erronées. Les modèles spécifiques sont principalement utilisés par des approches de type *fuzzing*. Par exemple, cette spécification peut être obtenue à partir de la documentation d'un protocole de communication [COMR] ou d'un format de fichier [FFUZZ]. Cette connaissance supplémentaire permet ainsi à ce type d'approche d'ajouter des erreurs à tous les niveaux des jeux de données. Cela permet donc de générer des jeux de données couvrant un large spectre des erreurs possibles et d'atteindre des portions de code du programme en cours de test que n'atteignent pas les mécanismes aveugles.

À noter que dans le cas d'un audit de sécurité, les erreurs introduites par ce type de mécanisme peuvent être générées de manière aléatoires mais aussi à l'aide de vecteurs d'attaques [OWASP]. C'est par exemple le cas de l'injection d'attaque réseau proposée par Neves et al. [NAC+06]. Le mécanisme proposé repose sur deux informations : la connaissance du protocole réseau utilisé par l'application que l'on désire tester et un vecteur d'attaques propre à la famille de protocole auquel celui-ci appartient.

Bien sûr, la contre-partie de disposer d'autant d'information est que cela augmente grandement le temps demandé pour le déploiement du mécanisme d'injection de fautes. En effet, il est nécessaire de renseigner ces informations dans la configuration de la plateforme [SPIKE, PEACH]. Cela implique que celles-ci doivent être correctement formatées pour pouvoir être utilisées par le mécanisme de génération

des jeux de données erronées. Le plus souvent cette tâche ne peut être faite que manuellement. On peut noter qu'il existe cependant des outils de *fuzzing* qui essaient d'automatiser cette tâche ou tout du moins de la faciliter en fournissant des mécanismes de reconnaissance automatique des protocoles [ADAFE].

1.4.2.3 Modèle en mémoire

Il arrive que la spécification d'un programme ne soit pas connue. Par exemple dans le cas d'une application fermée ou ne disposant pas d'une documentation librement accessible. Pour répondre à ce problème, plutôt que de revenir à des mécanismes aveugles, des approches dites en mémoire ont été proposées. Ce type d'approche permet, tout comme les modèles spécifiques, d'atteindre des états du programme en cours de test que n'atteignent pas les mécanismes aveugles, mais sans pour autant avoir connaissance de sa spécification. À noter que ce type d'approche permet également de répondre au problème d'automatisation du traitement de la spécification posé par les modèles spécifiques.

Pour cela, les mécanismes d'injection en mémoire cherchent à déplacer le problème de l'injection des erreurs, des points entrée du programme aux données directement manipulées en mémoire par ce dernier. Cela signifie qu'à l'instar des mécanismes logiciels de simulation de faute matérielle (voir section 1.4.2.1), les mécanismes d'injection en mémoire doivent avoir un accès direct à l'espace mémoire du processus de l'application. Cependant, le modèle de faute à simuler étant bien plus élaboré qu'une simple corruption aléatoire de la mémoire, le mécanisme d'injection doit pour être pertinent extraire un certain nombre d'informations du code exécuté par le processus. Ce modèle s'apparente donc plus à une approche en boîte grise.

Une première possibilité pour réaliser cela est l'interception des appels système [DM98, DM00]. Les erreurs peuvent alors être introduites directement dans les données retournées par ces appels système. Cette approche, bien que finalement proche des mécanismes de rejeu de données mutées (voir section 1.4.2.1), peut mettre en évidence l'existence d'un plus grand nombre d'erreurs de conception. En effet, grâce à l'apport d'informations sur le fonctionnement interne du programme, à savoir les appels système ainsi que les données qu'ils retournent, la génération des données erronées peut cibler un plus grand nombre de points d'entrée (par exemple, un fichier de configuration). Cependant, ce type de mécanisme reste encore limité face à certains types de données. C'est par exemple le cas des fichiers compressés ou des connexions réseaux chiffrées.

Pour répondre à ce problème, il est nécessaire d'être capable de modifier des données plus en profondeur dans la pile d'appels menant aux appels système. Pour réaliser cela, il est possible de reposer sur des outils de couverture de code. Ces outils sont originalement utilisés dans la phase de développement d'une application afin de s'assurer que celle-ci ne contient pas de code mort et donc que toutes ses branches sont exécutables. Dans le cas d'une campagne d'injections en mémoire,

ce type d'outil permet d'identifier la trace d'exécution des fonctions internes pour chaque appel système effectué par le programme [EFS]. Ces outils de couverture de code reposent sur des débogueurs ou des bibliothèques d'instrumentation, ils sont donc utilisés par la suite pour procéder à l'injection des données erronées. À noter que dans le cas d'un audit de sécurité, afin de se concentrer sur les fautes potentiellement capables de générer une intrusion, il est possible de donner la priorité, via un mécanisme de retour, aux jeux de données permettant d'atteindre des appels système potentiellement dangereux [ADAFE].

1.4.3 Positionnement des travaux

Dans cette thèse, nous voulons évaluer un mécanisme de détection des intrusions de niveau applicatif. Or nous avons vu à la section 1.3.3 que l'attaque à l'origine d'une intrusion peut être vue comme faute au sein du programme. Nous proposons donc de simuler le résultat d'une attaque en forçant le processus dans un état erroné. Pour cela, nos travaux reposent sur l'utilisation d'un mécanisme d'injection de fautes. Cependant, afin de reproduire les possibles états erronés d'un programme provoqués par une attaque, le mécanisme d'injection que nous proposons est capable de cibler les données qui influencent l'exécution des appels système.

Dans ce but, à la manière des mécanismes de simulation logicielle d'erreurs matérielles au niveau de la mémoire, notre mécanisme d'injection agit directement dans l'espace mémoire du processus. Toutefois, la simulation d'une attaque contre une application requiert un modèle de faute bien plus compliqué que la simple simulation d'une erreur au niveau de la mémoire. De plus, notre objectif étant de simuler un type d'attaque bien particulier, les modèles de fautes au niveau des points d'entrée du programme ne sont pas non plus suffisants. C'est pourquoi, à l'instar des mécanismes d'injection en mémoire, nous injectons les erreurs directement dans les données manipulées par le programme.

Cependant, tout comme ces derniers, nous avons besoin d'informations sur le fonctionnement interne du programme pour construire notre modèle de faute. C'est ce modèle qui permet à notre mécanisme d'injection de déterminer quels emplacements mémoire doivent subir une injection et à quel moment celle-ci doit avoir lieu. Toutefois, contrairement aux mécanismes d'injection en mémoire, nous ne nous limitons pas à l'interception des appels système ou à l'utilisation d'outils de couverture de code pour construire notre modèle. Nous proposons une approche par boîte blanche, reposant sur l'analyse statique du code source du programme à tester, pour déterminer les données qui pourraient être la cible d'une attaque.

De plus, afin d'améliorer la pertinence de notre mécanisme d'évaluation, nous proposons de déterminer quels états erronés parmi ceux que nous avons injectés sont les plus susceptibles d'être représentatifs d'une véritable intrusion. Une intrusion n'étant effective que lorsque des appels système illégaux ont été exécutés, nous utilisons pour cela une méthode d'apprentissage du comportement d'un programme

au niveau de ses appels système. Le modèle de comportement appris tient compte à la fois des séquences des appels systèmes mais aussi des arguments qui leurs sont passés en paramètre. Ce type de modélisation nous permet de déterminer quels états erronés ont effectivement permis de faire exécuter au programme des appels système illégaux.

Chapitre 2

Modèle de détection d'intrusion

Pour détecter une intrusion au sein d'un programme, nous proposons dans ces travaux de thèse une approche de type comportementale. Située au niveau applicatif, ce type d'approche requiert donc la construction préalable d'un modèle du comportement normal des programmes à surveiller et ceci afin de pouvoir mettre en évidence, pour chacun d'eux, les éventuelles déviations comportementales engendrées par une intrusion ou une tentative d'intrusion. Dans le cadre de l'approche que nous proposons, la construction de ce modèle est effectuée à partir du code source des applications concernées et celle-ci repose sur des techniques issues du domaine de l'analyse statique.

Notons que les techniques d'analyse statique sur lesquelles nous avons choisi de nous reposer nous assurent que les résultats obtenus sont des sur-approximations. Cette méthode de construction permet donc de générer un modèle qui est complet, c'est-à-dire un modèle qui inclut tous les comportements possibles du programme. Par conséquent, notre approche ne peut pas générer de fausse alerte. Cependant, si le modèle ainsi obtenu est complet celui-ci n'est pas correct. En effet, ce dernier représente une sur-approximation du comportement normal des programmes considérés. Par conséquent, il y a un risque avec notre approche que certaines intrusions ne soient pas détectées.

Dans ce chapitre, nous présentons l'approche comportementale que nous proposons pour détecter une intrusion dans un programme. Pour cela, nous abordons d'abord les différents types d'attaques dont peut être la cible un programme particulier et parmi celles-ci lesquelles notre approche cherche spécifiquement à détecter. Nous abordons ensuite en détails la méthode que nous proposons pour détecter le type d'attaques que nous avons identifié comme étant la cible de notre approche. Nous verrons notamment comment nous pouvons construire le modèle de comportement normal associé à notre approche pour la détection. Enfin, nous présentons aussi

l'outil que nous avons développé pour valider et tester notre approche ainsi que les outils d'analyse statique sur lesquels nous avons fondé notre implémentation.

2.1 Les attaques contre les applications

Une intrusion est le résultat d'une attaque réalisée avec succès sur le système d'information ciblé. Si l'attaque échoue, on dit alors qu'une tentative d'intrusion a eu lieu. Au niveau d'un hôte appartenant au système d'information, une attaque peut viser son système d'exploitation ou bien les applications qui s'exécutent au dessus de ce dernier. Quelle que soit la cible, une attaque peut chercher soit à épuiser les ressources du composant ciblé soit à exploiter une vulnérabilité dans sa conception. Dans le premier cas, l'attaque a nécessairement pour but de violer la propriété de disponibilité (on parle alors d'attaque par déni de service [Gli84, Ove99, Cri00]). Au niveau du système d'exploitation, un déluge de paquet réseau est un exemple de déni de service par épuisement de ressource [Edd07].

Dans le second cas, l'attaque cherche en priorité à violer les contraintes de confidentialité et d'intégrité. Au niveau du système d'exploitation, la corruption des données d'un pilote de périphérique mal conçu est un exemple d'exploitation de vulnérabilité permettant de violer l'intégrité du système [But07, Bul07]. Dans le cas où un utilisateur malveillant cherche à attenter à l'intégrité d'une application, l'attaque peut être réalisée à un moment donné en corrompant une ou plusieurs données dans l'espace mémoire du processus en cours d'exécution. Ceci est vrai quelle que soit la vulnérabilité exploitée par l'attaque : dépassement de tampon sur la pile ou sur le tas, dépassement de la capacité d'un entier, dépassement de chaîne formatée, etc.

Ces données peuvent être classées en deux catégorie : les données de contrôle et les données de calcul. Une attaque contre les données de contrôle cherche à corrompre les données utilisées par le processus pour contrôler son flot d'exécution. Une adresse de retour sur la pile ou un pointeur de fonction dans le gestionnaire d'exception sont des exemples de données prises pour cible par de telles attaques. En modifiant ces données, il est possible de forcer l'exécution du processus vers du code injecté ou hors contexte. Dans les deux cas, le code exécuté est illégal et se trouve sur un chemin invalide.

Une attaque contre les données de calcul cherche à corrompre les données utilisées par le processus pour effectuer les tâches qui lui incombent. Les variables contenant les valeurs nécessaires à l'évaluation des branchements conditionnels ou au calcul des arguments d'un appel système sont des exemples de données prises pour cible par de telles attaques. En modifiant ces données, il est possible d'utiliser de manière invalide du code légalement exécutable. Par exemple, il est possible d'effectuer un appel système illégal bien que ce dernier soit situé dans du code valide en l'exécutant au travers d'un chemin invalide ou en lui fournissant des arguments incorrects. Nous allons détailler pour ces deux types d'attaques différentes techniques d'exploitation.

2.1.1 Les attaques ciblant les données de contrôle

Ce type d'attaque cherche à corrompre les données utilisées par le système pour contrôler le flot d'exécution de l'application. La corruption de ce type de données peut permettre de détourner le flot d'exécution d'un processus vers des emplacements mémoire contenant du code qui ne peut en aucune circonstance être valide à exécuter. Le code exécuté est donc illégal et les appels systèmes effectués par ce dernier sont donc bien sûr tout aussi illégaux.

Il peut s'agir de code injecté, c'est-à-dire de code introduit en mémoire par l'attaquant, ou bien de code hors contexte, c'est-à-dire de code présent dans l'espace mémoire du processus mais hors contexte vis-à-vis de l'application (par exemple, le code des fonctions inutilisées d'une bibliothèque de fonctions chargée en mémoire par le programme). Dans la suite de cette section, nous allons détailler certaines méthodes permettant ces deux types d'exploitation.

2.1.1.1 Exécution de code injecté

Le détail des techniques permettant l'injection de code exécutable au sein d'un programme en cours d'exécution dépend fortement de l'architecture du processeur sur lequel le programme vulnérable est exécuté. Toutefois, le principe général varie peu, notamment pour l'exploitation d'un dépassement de tampon sur la pile. En effet, chaque processus possède une pile d'exécution qui contient les variables locales et les arguments passés en paramètre lors des appels de fonction, mais aussi un certain nombre de données utilisées par le système pour contrôler le flot d'exécution du processus considéré.

C'est notamment le cas des adresses de retour suite à l'appel d'une fonction ou encore des adresses des fonctions du gestionnaire d'exception. Lorsque l'instruction de retour de la fonction est exécutée ou bien qu'une exception est levée, ce sont ces adresses mémoire contenues sur la pile qui sont utilisées comme adresse de destination lors de la redirection du flot d'exécution à la fin de l'appel de fonction. Or les données reçues par le programme sont placées dans des tampons alloués préalablement qui sont eux aussi situés sur la pile du processus (à proximité de l'adresse de retour).

Si par exemple le programme n'est pas correctement conçu et qu'il ne vérifie pas que la taille des données reçues ne dépasse pas la taille maximale du tampon utilisé pour les contenir, alors un utilisateur malveillant aura la possibilité de modifier les adresses mémoire situées sur la pile. Si parmi les données envoyées par l'attaquant se trouve du code exécutable, il pourra alors le faire exécuter par la machine ciblée en s'assurant que l'adresse de retour une fois modifiée correspond bien à l'adresse de son code sur la pile. La figure 2.1 illustre les différents états de la mémoire du processus lors d'une attaque de type corruption de l'adresse de retour. On y voit successivement l'état de la mémoire avant l'appel de fonction, après l'appel de fonction et après l'attaque.

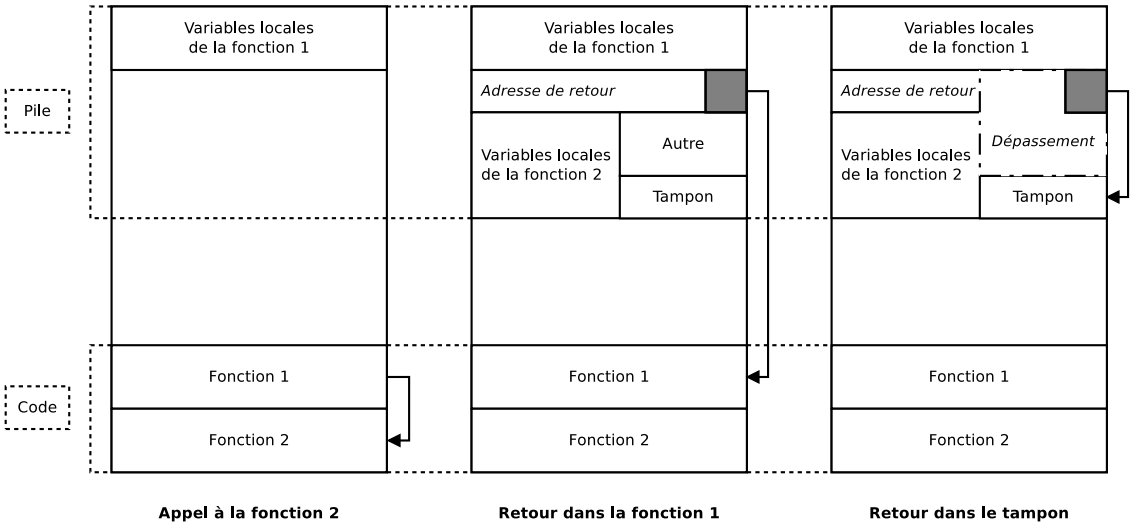


FIGURE 2.1 – Corruption de l'adresse de retour

2.1.1.2 Exécution de code hors contexte

Lorsque le système d'information ciblé possède un mécanisme qui détecte ou prévient l'exécution de code injecté (par exemple par la protection en exécution de la pile), un attaquant peut chercher à contourner ce mécanisme en détournant le flot d'exécution vers du code déjà présent en mémoire mais néanmoins utilisable pour réussir une attaque.

La technique consiste en l'injection non pas de code exécutable malveillant mais d'arguments malveillants qui seront passés en paramètre d'un ou plusieurs appels de fonction chargées en mémoire par les bibliothèques de fonctions partagées. En effet, même si le programme ne fait appel qu'à un petit nombre de fonctions d'une bibliothèque donnée, c'est toute la bibliothèque qui est chargée en mémoire lors du lancement du programme et l'ensemble de ses fonctions sont potentiellement activables par un attaquant.

Sur les système de type *UNIX*, il existe une bibliothèque de fonctions partagée qui est toujours présente dans l'espace mémoire du processus quelle que soit l'application exécutée, c'est la bibliothèque des fonctions standards du C (*libc*). De plus, cette dernière possède une fonction permettant d'exécuter des commandes sur le système ciblé à l'aide d'un seul argument, la fonction *system*. C'est donc bien souvent vers cette bibliothèque qu'un attaquant va chercher à détourner le flot d'exécution.

La réalisation d'une attaque via un retour dans une bibliothèque de fonctions partagée est très proche de la réalisation d'une attaque par injection de code. En effet, dans les deux cas on modifie une donnée de contrôle pour forcer la valeur du pointeur de code. Par exemple, dans le cas d'un dépassement de tampon sur la pile permettant de modifier l'adresse de retour, un attaquant peut forcer la fonction vulnérable à retourner au point d'entrée de la fonction *system* dans la bibliothèque

partagée *libc*. Cette fois, ce qui est contenu par la pile après l'adresse de retour, ce n'est pas du code machine comme dans la figure 2.1, mais la chaîne de caractère à passer en paramètre de la fonction *system*. On peut par exemple lui fournir comme argument la chaîne de caractère */bin/sh* pour obtenir illégalement un accès distant à la machine.

2.1.2 Les attaques ciblant les données de calcul

Ce type d'attaque cherche à corrompre les données utilisées par le processus pour effectuer ses calculs. La corruption de ce type de données peut permettre de détourner l'utilisation de code exécutable valide pour lui faire effectuer des appels système illégaux. Un attaquant peut par exemple chercher à corrompre des données utilisées dans le calcul des arguments passés en paramètre des appels système ou bien utilisées dans le calcul des évaluations conditionnelles qui mènent à l'exécution de ces appels. Il est bien sûr envisageable de modifier le comportement d'un processus en combinant ces deux types de corruption.

L'étude de ce type bien particulier d'attaque a été le sujet d'un article séminal dû à Chen et al. [CXS+05]. Dans ces travaux, plusieurs points sont mis en évidence à propos de ce type d'attaque. Tout d'abord, puisque ce type d'attaque n'injecte pas de code malveillant, mais cherche à détourner de manière malveillante du code valide, il nécessite une connaissance plus fine du fonctionnement interne des programmes ciblés. Parmi les mécanismes de détection et de prévention des intrusions les plus déployés, un grand nombre d'entre eux ne sont efficaces que contre les attaques ciblant les données de contrôle.

C'est pourquoi, malgré la plus grande complexité de réalisation que requièrent les attaques ciblant les données de calcul, ces dernières pourraient être utilisées par un attaquant pour contourner les mécanismes de détection en place sur sa cible. De plus, ces travaux montrent également que des vulnérabilités qui jusque là ont été exploitées uniquement pour réaliser des attaques contre les données de contrôle, peuvent aussi être exploitées pour réaliser des attaques contre les données de calcul. Enfin, ces travaux ont également montré que dans ce cas, la sévérité des compromissions obtenues peut être équivalente à celle résultant d'attaques contre les données de contrôle. Au final, l'ensemble de ces points suggère que ce type d'attaque représente bien une menace réelle.

Le nombre de données de calcul est grand au sein d'un programme et l'analyse complexe que requiert ce type d'exploitation peut difficilement être conduite pour chacune d'entre elles. Un utilisateur malveillant va donc utiliser la connaissance générale qu'il a du programme pour analyser finement en priorité les données qui contiennent les informations les plus susceptibles d'être utiles pour détourner le comportement de l'application. Nous allons maintenant détailler certaines informations que pourrait ainsi chercher à corrompre ce type d'exploitation.

2.1.2.1 Corruption des informations d'authentification

Les applications de type serveur, dans de nombreux cas, imposent à leurs utilisateurs de s'authentifier. Une application de ce type a donc besoin, lorsqu'elle reçoit une demande d'authentification, de charger en mémoire les données correspondant à l'utilisateur telles que son identifiant ou son mot de passe. Ces données sont alors conservées en mémoire jusqu'à ce que la phase d'authentification soit validée ou au contraire échoue.

La corruption de ces données durant l'exécution du programme peut permettre à un utilisateur malveillant de réussir la phase d'authentification ou d'usurper l'identité d'un autre utilisateur. Par exemple, dans le cas du serveur *OpenSSH* [OSSH] (pour les versions 2.2 ou antérieures), ce type de corruption donne à un attaquant la possibilité de forcer à vide le mot de passe de l'utilisateur ou encore de forcer l'identifiant (*uid*) de l'utilisateur à 0 (*root*).

2.1.2.2 Corruption des informations de configuration

L'utilisation de fichiers de configuration qui leurs sont propres est très répandue parmi les applications réseaux. Par exemple, ils sont utilisés par le serveur *Apache* [ASF] pour spécifier l'emplacement des interpréteurs de script, ou encore par le serveur *Netkit Telnetd* pour autoriser ou non le compte *root* à s'authentifier à distance. Un grand nombre de paramètres similaires peuvent être configurés de la sorte pour de nombreuses applications. Le plus souvent, ce type de fichier n'est lu qu'une fois, à l'initialisation du serveur. Les informations qui y sont contenues, une fois chargées en mémoire, sont conservées durant toute la durée de vie des processus et ne changent pas. Ces données en mémoire sont alors utilisées par le programme pour faire en sorte que durant son exécution son comportement est en conformité avec le fichier de configuration.

La corruption de ces données donne à un attaquant la capacité de modifier la configuration de l'application et donc la capacité d'altérer son comportement. Par exemple, dans le cas du serveur *Apache [ASF]*, en modifiant le chemin d'accès qui spécifie l'un des interpréteurs de scripts, il est possible de faire exécuter à l'application n'importe quel binaire exécutable présent sur le système. Si le serveur, au travers de ses pages *web*, donne à ses utilisateurs la possibilité d'envoyer des fichiers personnels, alors l'attaquant a même la possibilité de fournir ses propres binaires pour l'attaque.

Un autre exemple concerne le cas du serveur *OpenSSH* [OSSH] (mêmes versions que précédemment). Posons à nouveau l'hypothèse vu précédemment, à savoir qu'une vulnérabilité permet de s'authentifier sur la machine ciblée avec n'importe quel compte connu sans en détenir le mot de passe. Si la configuration du serveur ne permet pas au compte *root* de s'authentifier à distance, alors la gravité de l'intrusion sera moindre puisque l'utilisateur malveillant ne pourra avoir que des droits limités. Cependant, si cette donnée de configuration peut à son tour être corrompue,

il est alors possible de contourner cette limitation et donc d'augmenter le niveau de gravité de l'intrusion.

En effet, si le compte *root* n'est pas autorisé à s'authentifier à distance, l'attaquant peut tout d'abord exploiter la vulnérabilité pour modifier la variable utilisée pour contrôler cette option de configuration. Il lui suffit pour cela de corrompre les données qui correspondent à la configuration chargée en mémoire par le programme de la même manière qu'il peut corrompre les données qui correspondent à l'utilisateur avec lequel il cherche à s'authentifier. Ceci peut se faire avec n'importe qu'elle compte existant autre que *root*. Il ne lui reste plus qu'à réaliser une seconde exploitation de la vulnérabilité, celle qui permet de s'authentifier sans connaître le mot de passe correct. Au final, grâce à cette réalisation en deux temps de l'attaque, le niveau de compromission de l'intrusion reste le même malgré la mesure préventive qui consiste à interdire l'accès distant au compte *root*.

2.1.2.3 Corruption des informations de vérification

Une application, afin de garantir son intégrité, ne doit pas faire confiance à l'émetteur des données qu'elle reçoit, mais au contraire vérifier que ces dernières correspondent bien au format de données attendu. Ce type de vérification peut être contourné de deux manières. Tout d'abord, posons l'hypothèse que l'application considérée est sujette à une vulnérabilité qui ne permet de corrompre qu'un nombre restreint de données. Il suffit que parmi celles-ci certaines soient utilisées dans la vérification des données d'entrées pour donner à un utilisateur malveillant la possibilité d'injecter des données invalides et potentiellement de modifier le comportement du programme.

Une autre manière de contourner le mécanisme de vérification est de provoquer une situation de compétition (*race condition*) de type *TOCTTOU* (*Time Of Check To Time Of Use*) [BDBD96]. Cela est possible si la vulnérabilité est située dans le programme entre le moment de la vérification des données et le moment où ces données sont utilisées. Un attaquant peut alors fournir en entrée des données qui passent l'étape de la vérification, puis, avant que ces dernières ne soient utilisées, les altérer afin d'injecter des données invalides et donc potentiellement modifier le comportement du programme.

Prenons par exemple le cas d'une application qui utilise pour sa phase d'authentification une base de données *SQL*. Afin de se prémunir contre une attaque qui injecterait du code *SQL* [SQLI], l'application en question doit prendre certaines précautions quand elle reçoit des données de l'utilisateur. Par exemple, avant d'exécuter la phase d'authentification, l'application prend le soin de dupliquer ces données en échappant les caractères spéciaux d'une commande *SQL*. En remplaçant, avant l'authentification, la donnée qui contient l'adresse des données nettoyées par celle des données originelles, un attaquant peut contourner la vérification et donc réussir à injecter du code *SQL*.

2.1.2.4 Corruption des informations de branchement

Les applications utilisent des évaluations conditionnelles (dont les résultats sont des booléens), pour décider des branchements à effectuer dans le code durant l'exécution. Si une application présente une phase d'authentification, alors cette phase possède nécessairement une ou plusieurs évaluations conditionnelles afin de déterminer si l'accès doit être autorisé ou non. La corruption des informations utilisées par ces évaluations conditionnelles peut permettre à un attaquant de s'authentifier de manière frauduleuse auprès de l'application. De manière générale, ces évaluations influencent la possibilité d'exécuter ou non des appels système mais aussi la valeur des paramètres de ces appels système.

Prenons à nouveau pour exemple le cas du serveur *OpenSSH* [OSSH]. Lorsque celui-ci reçoit une demande d'authentification, il commence par charger les données de l'utilisateur correspondant. Si le mot de passe de l'utilisateur est vide, alors la demande d'authentification est immédiatement validée. Dans le cas contraire, le serveur exécute une boucle qui ne se terminera qu'une fois le bon mot de passe fourni par l'utilisateur. Supposons qu'une vulnérabilité existe entre le chargement des données de l'utilisateur et le test s'assurant que le mot de passe fourni est correct. Imaginons également que celle-ci permette de corrompre n'importe quelles données du programme. Il est donc possible d'attaquer ce dernier en forçant le mot de passe à vide au sein des données chargées pour l'utilisateur qui réalise la demande d'authentification. Si l'attaque est réalisée avant le test qui détermine l'entrée dans la boucle d'authentification, alors l'utilisateur malveillant peut directement s'authentifier. Sinon, il lui suffit de répondre par une chaîne de caractère vide à la demande de vérification de la boucle d'authentification.

Dans les deux cas, l'attaquant est capable de forcer l'authentification bien que le mot de passe ne soit pas vide et qu'il ne le connaisse pas. Si la vulnérabilité est spécifiquement située dans la boucle de vérification du mot de passe reçu, il est possible de l'exploiter autrement pour s'authentifier illégalement sur le système. En effet, un attaquant peut chercher à forcer la validité de l'évaluation de la condition de sortie de la boucle alors qu'il n'a à aucun moment fourni le mot de passe correct. À nouveau, un utilisateur malveillant peut donc obtenir de manière frauduleuse accès au serveur. Nous détaillerons le cas du serveur *OpenSSH* [OSSH] dans la section 2.3 pour illustrer l'approche que nous proposons.

2.2 Un modèle de détection déduit du code source

Contrairement aux attaques contre les données de contrôle dont le but est de faire exécuter au programme vulnérable du code invalide, les attaques contre les données de calcul cherchent à utiliser de manière illégale du code valide par la corruption des données qu'il manipule. Nous présentons ici le modèle orienté autour des variables que nous proposons afin de détecter ce type d'attaque.

Pour cela, nous verrons que la logique de Hoare [Hoa69], qui permet de raisonner sur la correction des programmes informatiques, peut également être utilisée pour détecter des attaques contre les données de calcul. C'est pourquoi nous commençons par présenter la logique de Hoare avant d'expliquer à l'aide d'un exemple comment se déroule une attaque contre les données de calcul et comment il est envisageable de la détecter à l'aide de cette logique. Nous présentons ensuite sur un autre exemple le modèle que nous proposons dans ces travaux pour effectivement détecter ce type d'attaque ainsi qu'une méthode pour construire ce modèle de manière automatisée grâce à l'analyse statique.

2.2.1 La logique de Hoare

Un programme est considéré correct s'il effectue sans erreur et ce dans les situations spécifiées les tâches qui lui sont confiées. Afin de pouvoir vérifier cela il est nécessaire de décrire précisément et sans ambiguïté l'ensemble de ces tâches ainsi que ce que ces dernières sont autorisées ou non à faire. C'est cette description qui constitue la spécification du programme.

Idéalement, le programme est la traduction exacte de cette spécification dans un langage de programmation. En pratique, il s'agit souvent d'une approximation de celle-ci. En effet, de nombreux programmes déployés sont en réalité incorrects car il subsiste des situations rares mais possibles où l'exécution des tâches provoquent des erreurs. C'est également le cas s'il existe des situations possibles mais non prévues par la spécification car le programme ne vérifie pas que les tâches qui s'exécutent sont strictement limitées à ce qu'elles sont autorisées à faire.

La logique de Hoare [Hoa69] a pour but de formaliser la relation entre langage de programmation et langage de spécification afin de permettre de raisonner sur la correction des programmes informatiques. L'utilisation de la logique de Hoare permet alors pour un programme donné d'obtenir une preuve de programme et donc de vérifier à l'aide de sa spécification que ce dernier se comporte bien comme il est censé le faire.

2.2.1.1 Les triplets de Hoare

Pour cela, la logique de Hoare considère que le programme est un transformateur d'états. En effet, chaque instruction exécutable du programme considéré fait évoluer son état, à savoir la valeur de l'ensemble constitué de toutes les variables du programme. La logique de Hoare exprime donc pour chaque évolution de cet état, des propriétés sur l'état final à partir des propriétés de l'état initial. Chacune de ses propriétés ne portent que sur les variables du programme que modifie l'instruction exécutée correspondante. La formulation de l'ensemble de ces propriétés constitue la spécification exacte implémentée par le programme (et qui peut différer de la spécification théorique).

L'expression de ces propriétés est fondée sur la notion de triplet de Hoare, de la forme $\{P\}\ C\ \{Q\}$. Un triplet de Hoare est composé de l'instruction C qui fait évoluer l'état du programme, de la pré-condition $\{P\}$ et de la post-condition $\{Q\}$. Plus précisément, $\{P\}$ et $\{Q\}$ sont des assertions, c'est-à-dire des formules logiques du langage de spécification représentant des propriétés sur les variables du programme. Le plus souvent, ces formules sont exprimées à l'aide de la logique des prédicats. La notation $\{P\}\ C\ \{Q\}$ exprime donc que suite à l'exécution de l'instruction C, l'état du programme vérifie la propriété exprimée par $\{Q\}$ si l'état initial du programme vérifiait la propriété exprimée par $\{P\}$.

2.2.1.2 Les règles d'inférence

Pour le cas de la programmation impérative, la logique de Hoare définit des règles d'inférence pour toutes les instructions de base de cette famille de langage. Les notations utilisées pour exprimer ces règles d'inférence sont les suivantes :

- La négation de P est notée $\neg P$.
- La conjonction de P et de Q est notée $P \wedge Q$.
- La disjonction de P et de Q est notée $P \vee Q$.
- L'implication de Q par P est notée $P \Rightarrow Q$.
- $P[x \leftarrow E]$ désigne l'expression P dans laquelle les occurrences de la variable x ont été remplacées par l'expression E.

Une règle d'inférence est une fraction de la forme $\frac{P_0,\ldots,P_n}{C}$ où le numérateur est un n-uplet de prémisses et le dénominateur une conclusion. Une règle d'inférence donnée exprime donc la relation de déduction liant la conclusion à l'ensemble des prémisses correspondantes. Une règle d'inférence qui ne possède aucun prémisse est appelée axiome. Dans le cas de la logique de Hoare, les prémisses et la conclusion sont exprimées à l'aide de triplet de Hoare. Ce type de règles d'inférence exprime donc le fait que si l'ensemble des prémisses sont vraies, alors la conclusion est vraie. Les règles d'inférence utilisées par la logique de Hoare pour modéliser l'évolution des états sont les suivantes :

–
$$\frac{}{\{P\}\ \textbf{skip}\ \{P\}}$$
Il s'agit de la règle vide. L'état du programme ne change pas, ce qui était vrai à l'état initial est toujours vrai à l'état final.

–
$$\frac{}{\{P[x \leftarrow E]\}\ x := E\ \{P\}}$$
Il s'agit de la règle d'affectation. Si la propriété était vraie à l'état initial pour $P[x \leftarrow E]$, alors elle est aussi vrai pour x à l'état final puisque $x := E$. Par exemple, $\{x + 1 = 3\}\ y := x + 1\ \{y = 3\}$.

–
$$\frac{\{P\}\ C_i\ \{T\}\ ,\ \{T\}\ C_{i+1}\ \{Q\}}{\{P\}\ C_i; C_{i+1}\ \{Q\}}$$

Il s'agit de la règle de composition. Elle permet de modéliser une séquence d'instructions. L'exécution séquentielle de C_i puis de C_{i+1} est notée $C_i; C_{i+1}$. par exemple, si on considère les deux triplets $\{x+1=3\}\ y := x+1\ \{y=3\}$ et $\{y=3\}\ z := y\ \{z=3\}$, alors on peut en conclure que $\{x+1=3\}\ y := x+1; z := y\ \{z=3\}$.

–
$$\frac{P' \Rightarrow\ P\ ,\ \{P\}\ C\ \{Q\}\ ,\ Q \Rightarrow\ Q'}{\{P'\}\ C\ \{Q'\}}$$

Il s'agit de la règle de conséquence. Elle permet de déduire des propriétés précédentes d'autres propriétés plus complexes. Prenons comme exemple le triplet suivant : $\{x+1=3\}\ y := x+1\ \{y=3\}$. Posons la relation d'équivalence suivante : $x+1=3 \Leftrightarrow x=2$. On peut alors en déduire que $\{x=2\}\ y := x+1\ \{y=3\}$.

–
$$\frac{\{B \wedge P\}\ C_T\ \{Q\}\ ,\ \{\neg B \wedge P\}\ C_F\ \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ C_T\ \textbf{else}\ C_F\ \textbf{endif}\ \{Q\}}$$

Il s'agit de la règle de condition. Dans la branche C_T la pré-condition P est vérifiée et B est vraie tandis que dans la branche C_F la pré-condition P est aussi vérifiée et B est faux. Par exemple, si on considère les deux triplets $\{true \wedge x=0\}\ y := x\ \{y=0\}$ et $\{true \wedge x \neq 0\}\ y := 0\ \{y=0\}$ alors on peut en conclure que $\{true\}$ **if** $x=0$ **then** $y := x$ **else** $y := 0\ \{y=0\}$.

–
$$\frac{\{P \wedge B\}\ C\ \{P\}}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ C\ \textbf{done}\ \{\neg B \wedge P\}}$$

Il s'agit de la règle de boucle. Lorsque C s'exécute, la précondition est vérifiée et B est vraie tandis que la post-condition implique la pré-condition et qu'en sortie B est faux. P est appelé invariant de boucle. Par exemple, si on considère le triplet suivant $\{x \geq 0 \wedge x < y\}\ x := x+1\ \{x \geq 0\}$, alors on peut en conclure que $\{x \geq 0\}$ **while** $x < y$ **do** $x := x+1$ **done** $\{x \geq 0 \wedge \neg(x < y)\}$.

2.2.2 Détection des attaques contre les données de calcul

Les attaques ciblant les données de calcul peuvent engendrer des compromissions aussi sévères que les attaques ciblant les données de contrôle pour divers schémas de vulnérabilités existants (voir section 2.1.2). Parmi les vulnérabilités utilisées en exemple, on trouve notamment une vulnérabilité qui touche le très déployé serveur *FTP* libre *WU-FTPD* [WUFTP] (pour les versions 2.6.0 ou antérieures). Cette faille est due à la portion de code responsable de la journalisation [C0133]. En effet, ce code ajoute des entrées dans le journal sans utiliser de chaînes de formatage. Là encore, il s'agit d'un schéma de vulnérabilité bien connu. Si une entrée contient des données contrôlables par le client, alors un utilisateur malveillant peut utiliser cela pour modifier directement la mémoire du processus.

2.2.2.1 Exploitation de la vulnérabilité

La figure 2.2 (colonne de gauche) est un exemple de code contenant une vulnérabilité similaire à celle décrite précédemment. Dans cet exemple, la vulnérabilité se situe à la ligne 10. Cette ligne de code affiche sans utiliser de chaînes de formatage l'entrée utilisateur reçue à la ligne 07. En conséquence, un utilisateur malveillant peut fournir en entrée du programme une chaîne de formatage lui permettant d'avoir un accès direct en écriture à la mémoire du processus. Dans ce cas précis, c'est la variable *uid* qui se trouve être une cible de choix.

```
01 int main(int argc, char **argv)
02 {
03   char buf[64];
04   uid_t uid;

05   uid = atoi(argv[1]);

06   seteuid(uid);

07   while(fgets(buf, 64, stdin))
08   {

09     seteuid(0);

10     printf(buf);

11     seteuid(uid);
12   }

13 }
```

```
01 int main(int argc, char **argv)
02 {
03   char buf[64];
04   uid_t uid;

     //@ {true}
05   uid = atoi(argv[1]);

     //@ {uid==atoi(argv[1])}
06   seteuid(uid);

     //@ {uid==atoi(argv[1])}
07   while(fgets(buf, 64, stdin))
08   {

     //@ {uid==atoi(argv[1])}
     //@   AND {s}
     //@   AND {s -> buf[0] != '\0'}
     //@ {uid==atoi(argv[1])}
     //@   AND {buf[0] != '\0'}
09     seteuid(0);

     //@ {uid==atoi(argv[1])}
     //@   AND {buf[0] != '\0'}
10     printf(buf);

     //@ {uid==atoi(argv[1])}
     //@   AND {buf[0] != '\0'}
11     seteuid(uid);
12   }

     //@ {uid==atoi(argv[1])}
     //@   AND {buf[0]=='\0'}
13 }
```

FIGURE 2.2 – Exemple d'annotations de Hoare dans un programme en langage *C* contenant une vulnérabilité de type *string format*

Effectivement, en forçant sa valeur à zéro par exemple (*root*), un attaquant peut élever son niveau de privilèges, sans corrompre le flot d'exécution, lors de l'appel à la fonction *seteuid* à la ligne 11. Cet exemple montre bien comment une attaque contre les données de calcul peut violer des contraintes que les variables devraient normalement respecter. Ainsi, la variable *uid* devrait être constante durant toute l'exécution de la boucle (de la ligne 07 à la ligne 12) et être égale à la valeur qui lui a été assignée à la ligne 05. Afin de pouvoir détecter ce type d'attaque à l'exécution du programme, nous souhaitons construire ce type de contraintes de manière automatique.

2.2.2.2 Application de la logique de Hoare

Nous allons maintenant appliquer la logique de Hoare à l'exemple précédent pour déterminer s'il est possible de détecter ce type d'attaque grâce au code source. L'analyse statique du code source d'un programme afin travailler avec des prédicats Hoare est une approche déjà utilisée par les outils de vérification de programme. Cependant, dans notre cas, nous ne pas voulons vérifier que le programme respecte bien les propriétés exprimées par des prédicats fournis en entrée. Ce que nous voulons, c'est déduire de manière automatique des propriétés.

Pour illustrer cette approche, nous avons construit les différents prédicats qui peuvent être déduits du code source en exemple dans la figure 2.2 (colonne de droite). Dans cet exemple, on suppose que l'appel à la fonction *printf* à la ligne 10 ne peut modifier aucune donnée interne au programme mais uniquement écrire sur la sortie standard. En reposant sur cette hypothèse, on obtient ainsi sur la valeur de la variable *uid*, utilisée à la ligne 11, la pré-condition suivant : $uid == atoi(argv[1])\ AND\ buf[0] !=' 0'$. Ce prédicat correspond exactement à la contrainte que nous souhaiterions vérifier à cet endroit précis du code afin de détecter une attaque sur la variable *uid*.

En effet, si nous vérifions durant l'exécution du programme que cette contrainte est bien respectée juste avant l'appel à la fonction *setuid* à la ligne 11, alors nous pouvons détecter une élévation de privilèges illégale par corruption de la variable *uid*. Cet exemple illustre bien le fait que des contraintes permettant de détecter des attaques contre les données de calcul peuvent être déduites du code source de l'application. Toutefois, les approches traditionnelles qui utilisent la logique de Hoare reposent sur la génération manuelle de prédicats qui sont ensuite vérifiés par un assistant de preuve.

De plus, il faut savoir que construire de manière automatique la spécification implémentée par un programme en utilisant les règles d'inférence de Hoare peut être très difficile voire impossible dans le cas général. C'est particulièrement le cas pour certains programmes complexes mettant en œuvre des règles d'implication complexes ou reposant sur des résultats mathématiques difficiles dont la démonstration est déjà hors de portée des assistants de vérification.

C'est pourquoi nous nous concentrons dans la section qui suit sur les techniques d'analyse statique qui sont utilisables pour construire de manière automatique notre modèle de comportement normal orienté autour des variables.

2.3 Un modèle orienté autour des variables

Nous avons vu que les données de calcul, qui par leur corruption permettent d'utiliser de manière illégale du code valide et donc de détourner le programme de son comportement normal, sont utilisées pour contenir la valeur des variables issues du code source. Le modèle que nous voulons construire repose sur la vérification, au sein du programme, des variables qui influencent l'exécution des appels système effectués par ce dernier.

Pour présenter et expliquer l'approche que nous proposons afin de détecter ce type d'attaque, nous allons détailler un autre cas issu des exemples de l'article de Chen et al [CXS+05]. Nous présentons d'abord comment il est possible de détecter des attaques qui ont réellement été mises en œuvre contre le programme vulnérable et quelles contraintes peuvent les détecter. Nous expliquons ensuite comment il est possible de construire de manière automatique le modèle de détection que nous proposons en traitant les problèmes d'analyse statique que pose ce modèle orienté autour des variables.

2.3.1 Présentation du modèle sur un exemple

Le très déployé serveur *SSH* libre *OpenSSH* [OSSH] fut sujet en 2001 (versions 2.2 ou antérieures) à une vulnérabilité de type dépassement de capacité d'un entier. En effet, dans une fonction utilisée par ce dernier pour calculer une somme de contrôle, un entier d'une taille de 2 octets reçoit comme valeur le résultat d'un calcul utilisant des données envoyées par le client et qui peut atteindre une taille de 4 octets [Zal01]. Le code de la figure 2.3 (colonne de gauche) est inspiré de cette version particulière de *OpenSSH* et reproduit la structure de base de la partie du véritable code qui nous intéresse. Nous utiliserons dans la suite de cette section cet exemple pour présenter et illustrer notre approche.

2.3.1.1 Description de la vulnérabilité

Le code de la figure 2.3 reproduit la dernière partie de la phase d'authentification du server *SSH*. Lorsqu'une demande d'authentification est reçue par le programme, celui-ci charge en mémoire la structure de données associée à l'utilisateur correspondant à la demande d'authentification. C'est cette structure qui contient entre autres son mot de passe et son *uid*. Si le mot de passe de l'utilisateur est vide, alors l'authentification est immédiatement validée par le programme (l'exécution du code passe directement la ligne 22).

```
00. void do_authentication()
01.{
02.   int ok = 0;
      ...

03.   if(!strcmp(pwd, ""))
04.     /* users with no password */
05.   else
06.     /* do_authloop(); */
07.     while(ok != 1) {
08.
09.       type = packet_read(buf);
10.
11.       switch (type) {
            ...

12.       case CMSG_AUTH_PWD:
13.
14.
15.         ok = auth_pwd(pwd, buf);
16.         break;
            ...

17.       }
18.     }
19.
20.
21.
22.   do_authenticated(user);
23. }
```

```
00. void do_authentication()
01. {
02.   int ok = 0;
      ...

03.   if(pwd[0] != '\0')
04.     /* users with no password */
05.   else
06.     /* do_authloop(); */
07.     while(ok != 1) {
08.
09.       type = packet_read(buf);
10.
11.       switch (type) {
            ...

12.       case CMSG_AUTH_PWD:
13.
14.         assert(pwd[0] != '\0');
15.         ok = auth_pwd(pwd, buf);
16.         break;
            ...

17.       }
18.     }
19.
20.   assert(ok==0 OR (ok==1
21.     AND type==CMSG_AUTH_PWD));
22.   do_authenticated(user);
23. }
```

FIGURE 2.3 – Exemple d'attaques contre les données de calcul inspiré de la version vulnérable de *OpenSSH*

Dans le cas contraire, le serveur va exécuter en boucle des instructions qui vont recevoir d'autres informations de la part de l'utilisateur distant. Ce qui nous intéresse dans cet exemple, c'est le cas où le serveur reçoit une demande d'authentification par mot de passe (il existe d'autres méthodes d'authentification, par clé *RSA* par exemple). Si le mot de passe fourni par l'utilisateur qui demande à s'authentifier est correct, alors l'exécution de la boucle se termine (et le programme exécute maintenant le code à partir de la ligne 22).

Il faut savoir que, durant cette phase d'authentification, la fonction responsable de la réception des données envoyées par l'utilisateur distant (appelée à la ligne 09 (*packet_read*)) fait appel à la fonction de calcul de somme de contrôle alors que cette dernière présente une vulnérabilité. Dans cette fonction vulnérable, la don-

née concernée par le dépassement d'entier est utilisée pour calculer un masque qui est lui-même utilisé pour empêcher une opération d'écriture de sortir des limites d'un tableau. Un utilisateur malveillant peut dès lors utiliser cette faute dans le programme pour modifier de manière non prévue le masque afin de pouvoir modifier des données à n'importe quel emplacement mémoire. Pekka et Kalle [PK02] fournissent une analyse détaillée de cette vulnérabilité.

2.3.1.2 Les scénarios d'attaque

Cette vulnérabilité peut par exemple être utilisée durant l'exécution de la boucle *do_ authloop* de manière à forcer la variable *pwd* à présenter comme valeur une chaîne de caractère vide. Ainsi, un utilisateur malveillant aura la possibilité de s'authentifier sur le système avec en se faisant passer pour l'utilisateur de son choix (par exemple, *root*) sans avoir à fournir le bon mot de passe. Une explication détaillée d'une telle exploitation de la faille a été réalisée par Starzetz [Sta01]. Notons que le nom d'utilisateur utilisé pour réaliser l'attaque doit nécessairement correspondre à un compte existant sur le système ciblé. En effet, un nom d'utilisateur inexistant ne permet pas d'atteindre cette portion de code.

Une autre manière permettant de forcer la réussite de la phase d'authentification est de forcer la valeur de la variable de boucle *ok*. Il s'agit de la variable qui permet de quitter la boucle d'authentification. Celle-ci ne peut normalement être égale à 1 que si l'authentification de l'utilisateur est réussie. Pour contourner cela, il faut que l'utilisateur malveillant envoie des informations dont le traitement ne modifie pas cette variable (c'est-à-dire tout sauf une demande d'authentification, on peut par exemple demander un nouvel échange de clés de chiffrement). Il est alors possible de forcer la valeur de *ok* à 1 en reposant sur la même vulnérabilité que celle utilisée précédemment. À nouveau, on peut ainsi s'authentifier sur le système avec n'importe quel compte connu.

2.3.1.3 La détection de l'intrusion

Cependant, dans les deux cas l'exploitation de la vulnérabilité met l'espace mémoire du processus dans un état incohérent. En effet, dans le cas de l'attaque qui modifie la variable *pwd*, si le mot de passe de l'utilisateur concerné était effectivement vide en premier lieu, alors jamais le serveur n'aurait dû exécuter la boucle d'authentification. Autrement dit, si le flot d'exécution atteint cette boucle d'authentification, alors le mot de passe ne peut pas être une chaîne de caractères vide. Si on vérifie cette contrainte avant l'utilisation de la variable *pwd* alors cette première attaque est détectée. Notons que la vérification est placée juste avant l'appel de fonction concernée. Ceci nous permet d'être sûr que si attaque il y a, alors celle-ci ne peut avoir lieu après la vérification (on suppose que le code qui effectue la vérification n'est pas attaquable).

Dans le cas de l'attaque qui modifie la variable de boucle *ok*, celle-ci ne peut valoir 1 que si le dernier message reçu est une demande d'authentification. Si on vérifie cette contrainte avant l'appel à la fonction qui autorise l'accès au système (*do_ authenticated*), alors cette seconde attaque est aussi détectée. À nouveau et pour les mêmes raisons, la vérification est placée juste avant l'appel de fonction concernée.

L'objectif de notre approche pour la détection est donc de déduire du code source ce type de contraintes puis de les traduire sous forme d'assertions exécutables à insérer dans ce code source. La figure 2.3 (colonne de droite) présente les assertions nécessaires à la détection de ces attaques dans le cas de notre exemple. Dans le cas de la première attaque, l'invariant à vérifier est le suivant : $pwd[0] != '\backslash 0'$. Dans le cas de la seconde attaque, l'invariant à vérifier est le suivant : $ok == 0\ OR\ (ok == 1\ AND\ type == CMSG\_AUTH\_PWD)$. Le modèle de détection que nous proposons dans ces travaux de thèse repose sur le calcul automatisé de ce type de contraintes.

2.3.1.4 Formalisation du modèle

Les données de calcul sont utilisées par le programme pour contenir la valeur des variables présentes dans son code source. Nous définissons l'état interne d'un programme comme l'ensemble des éléments mémoire utilisés pour contenir ces variables. Ces dernières peuvent être liées entre elles (par exemple, deux variables particulières doivent être égales), ou posséder des propriétés qui leur sont propres (par exemple, une variable ne peut prendre qu'un nombre restreint de valeurs). Ces propriétés sont des invariants et constituent la base de notre modèle. Notre objectif est de compléter les mécanismes de détection efficaces vis-à-vis des attaques contre les données de contrôle (basés sur la vérification des séquences d'appels système) en se concentrant sur l'exactitude des données de calcul. Notre approche consiste donc en la recherche d'invariants au sein d'un programme afin de détecter des corruptions de variables induites par une anomalie.

Cependant, contrairement aux autres méthodes et ce dans un souci de performance, nous ne voulons pas vérifier de telles propriétés en tout point du programme et pour l'ensemble des variables. Ce que nous voulons, c'est seulement les vérifier sur les chemins menant aux appels système et seulement pour le sous-ensemble de variables dont dépendent ces appels. Cet ensemble constitue en effet l'ensemble des cibles potentielles pour un attaquant. Un appel système dépend de deux types de variables : les variables utilisées pour calculer les arguments qui lui sont passés en paramètre et les variables utilisées pour atteindre cet appel (par exemple, les variables utilisées lors d'une évaluation conditionnelle sur un chemin menant à cet appel).

Nous pouvons donc définir pour un appel système donné AS_i son modèle de comportement par le triplet (AS_i, V_i, C_i) où V_i est l'ensemble des variables dont dépend

l'appel système et C_i l'ensemble des contraintes que ces dernières doivent respecter. Nous définissons alors le modèle de comportement normal du programme MCN comme l'ensemble des AS_i , soit $MCN = \{\forall i, (AS_i, V_i, C_i)\}$. Une attaque contre les données de calcul consiste alors à modifier la valeur d'une ou plusieurs variables appartenant à V_i. Une ou plusieurs des contraintes appartenant à C_i peuvent alors être violées. Un programme est donc considéré dans un état correct tant que toutes ses contraintes sont vérifiées durant son exécution. Une intrusion est détectée dès lors qu'au moins une seule des contraintes devient fausse.

Pour pouvoir construire de manière automatique un tel modèle, deux problèmes doivent être résolus par des techniques d'analyse statique : comment déterminer les variables sur lesquelles doivent porter les contrôles et comment déterminer les contraintes que doivent vérifier ces contrôles. Nous répondons à ces questions dans les deux sections suivantes.

2.3.2 Découverte des variables à surveiller

Nous allons aborder dans cette section la question des variables que l'on souhaite contrôler durant l'exécution du programme et pour chacune d'entre elles à quels moments il est utile d'effectuer un contrôle. En effet, comme nous allons le voir, les mécanismes de contrôle que nous proposons sont embarqués dans le programme et la manière dont on va déterminer les ensembles de variables dont on veut vérifier la valeur va dépendre de l'emplacement de ces contrôles dans le programme. Pour cela, nous expliquons d'abord quels sont les points du programme qu'il est utile de contrôler afin de pouvoir détecter des attaques contre les données de calcul. Puis nous présentons une méthode permettant de déterminer les ensembles de variables qui nous intéressent ainsi qu'une technique d'analyse statique particulière qui permet d'obtenir ce résultat.

2.3.2.1 Localisation des contrôles

Comme nous l'avons vu dans la section 2.1, l'objectif d'une attaque contre une application est d'exécuter sur le système ciblé des appels système illégaux. Dans le cas d'une attaque qui cible les données de calcul, celle-ci va chercher à détourner les appels système effectués par du code valide appartenant néanmoins au programme (voir la section 2.1.2). Idéalement, on voudrait donc pouvoir contrôler pour chaque appel système l'ensemble des variables dont il dépend. Seulement, cela n'est pas toujours possible.

En effet, les appels système ne sont pas nécessairement tous présents dans le code source de l'application. C'est par exemple le cas des appels système effectués au travers d'appels à des fonctions fournies par des bibliothèques de fonctions partagées. De plus, l'ensemble des variables dont dépend un appel système donné ne sont pas nécessairement toutes accessibles dans le code source au moment de l'exécution de

l'appel. C'est par exemple le cas des variables locales déclarées dans les fonctions situées en amont dans la pile d'appel.

Pour résoudre ces deux problèmes, il est nécessaire plutôt que de contrôler directement les appels système de distribuer ces contrôles tout au long du code source, sur les chemins d'exécution menant aux appels système. Pour distribuer ainsi les contrôles, nous avons choisi de contrôler les appels de fonction (y compris les appels système si cela est possible). De cette façon, l'ensemble des fonctions présentes dans la pile d'appel menant à un appel système sont contrôlées pour les appels présents dans le code source et pour chaque contrôle, seules les variables accessibles au moment de l'appel sont contrôlées.

2.3.2.2 La coupe de programme

Si on possède le code source d'un programme, alors il est possible, pour un appel de fonction donnée, de connaître l'ensemble des variables dont il dépend. Ceci peut notamment être obtenu grâce à la réalisation d'une coupe de programme. En analyse statique, on appelle coupe de programme le sous-ensemble du programme qui contient toutes les instructions dont dépend une instruction particulière choisie appelée point d'intérêt ou encore critère de coupe. Le résultat de cette coupe est un nouveau programme, sous-ensemble du programme d'origine, et qui se comporte exactement comme ce dernier pour un ensemble de scénarios d'exécution donnés et jusqu'au point d'intérêt choisi.

```
00. unsigned int i = 0;
01.
02. unsigned int somme = 0;
03. unsigned int produit = 1;
04.
05. unsigned int N = lire_entier();
06.
07. while(i != N)
08. {
09.   somme = somme + i;
10.   produit = produit * i;
11.
12.   i++;
13. }
14.
15. afficher_entier(somme);
16. afficher_entier(produit);
```

```
00. unsigned int i = 0;
01.
02. unsigned int somme = 0;
03.
04.
05. unsigned int N = lire_entier();
06.
07. while(i != N)
08. {
09.   somme = somme + i;
10.
11.
12.   i++;
13. }
14.
15. afficher_entier(somme);
16.
```

FIGURE 2.4 – Exemple d'une coupe de programme

Cependant, obtenir une coupe de programme minimale est un problème indécid-

able : c'est pourquoi les outils permettant de calculer de manière automatique des coupes de programme produisent des sur-approximations. Il est possible de calculer une coupe de programme avec pour critère de coupe plusieurs instructions. Il s'agit simplement de l'union des coupes de programme ayant pour point d'intérêt chacune de ces instructions.

L'utilisation de la coupe de programme permet par exemple de faciliter la localisation d'erreurs dans le code source. En effet, une fois que l'on connaît la portion de code à l'origine d'une défaillance, en calculant la coupe de programme ayant pour critère de coupe cette portion de code, le programmeur peut concentrer ses recherches sur un sous-ensemble d'instructions réduit mais qui contient toutefois nécessairement le code erroné. Le calcul de coupes de programme peut également être utilisé pour faciliter l'optimisation d'un programme ou de manière générale pour faciliter l'analyse d'un code source (dans les deux cas, en permettant au développeur de se concentrer sur une partie critique et ses dépendances). Dans ces travaux de thèse, cette méthode d'analyse statique est utilisée pour déterminer l'ensemble de variables dont dépend un appel de fonction donné.

La coupe de programme a été définie en 1981 par Weiser [Wei81] et depuis de nombreuses méthodes pour la calculer ont été proposées [Tip95]. La plupart sont des méthodes statiques, c'est-à-dire que la méthode de calcul d'une coupe de programme s'applique au code source et ne nécessite aucune autre information. Des variations de ces approches statiques ont été proposées, notamment pour répondre à des besoins spécifiques, comme la coupe de chemin [JM05] ou la coupe de programme préservant la confidentialité de l'information [MS08]. Cette dernière variation permet d'utiliser des coupes de programme pour faciliter la réalisation d'un audit de sécurité d'une application. Bogdan Korel and Janusz Laski ont également proposé une méthode dynamique de coupe de programme qui s'applique à une exécution spécifique du programme fournie sous forme de trace d'exécution [KL88]. Cette dernière approche permet de prendre en compte les valeurs de variables qui n'auraient pu être déterminées statiquement.

La figure 2.4 montre un exemple simple de coupe de programme. Le programme original est situé dans la colonne de gauche. Celui-ci reçoit une valeur entière N puis calcule et affiche la somme et le produit des N premiers nombres entiers. On choisit d'effectuer une coupe de programme avec pour point d'intérêt l'instruction ligne 15 (*afficher_ entier(somme) ;*). Le résultat est situé dans la colonne de droite. On obtient bien un programme réduit qui calcule et affiche maintenant uniquement la somme des N premiers nombres entiers. On note que l'instruction ligne 15 qui constitue le critère de coupe est présente dans la coupe de programme bien que celle-ci ne dépende pas d'elle-même.

Dans cet exemple, le point d'intérêt n'a pas été choisi au hasard. En effet, il s'agit d'un appel de fonction. Ce dont nous avons besoin pour construire notre modèle de détection, c'est de déterminer pour un appel de fonction donné, l'ensemble des variables dont il dépend, en valeur ou en contrôle. Or dans le cas de l'appel à la

fonction *afficher_ entier* à la ligne 15, il s'agit précisément de l'ensemble des variables utilisées par les instructions du sous-programme obtenu. Au final, l'ensemble des variables dont dépend un appel de fonction peut être déterminé en calculant l'ensemble des variables présentes dans la coupe de programme ayant pour point d'intérêt cet appel.

2.3.2.3 Le graphe de dépendance du programme

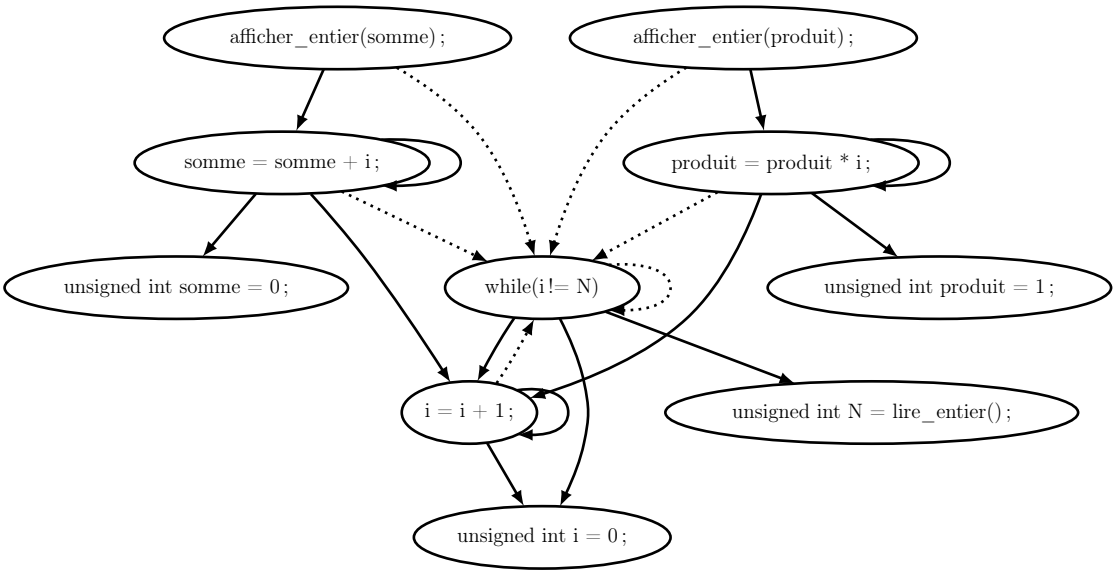


FIGURE 2.5 – Graphe de dépendance du programme

Le premier article parlant de coupe de programme [Wei81] propose une méthode de calcul reposant sur l'analyse du flot de contrôle du programme et des flots d'information dans ce programme. Une autre méthode proposée pour calculer une coupe de programme repose sur le calcul du graphe de dépendances du programme [KKP+81, FOW87] considéré. Il s'agit d'un graphe orienté dont les sommets représentent les instructions du programme et les contrôles de prédicat tandis que les arcs représentent les dépendances en contrôle et en valeur. Le calcul d'une coupe de programme peut alors être formulé sous la forme d'un problème d'accessibilité. Ce graphe permet également de connaître l'ensemble des variables dont dépend une instruction particulière et pour chacune d'entre elles s'il s'agit d'une dépendance en contrôle ou bien d'une dépendance en valeur.

Pour illustrer cette approche, nous avons créé le graphe de dépendances du programme en exemple dans la figure 2.4 (colonne de gauche). Le graphe est représenté dans la figure 2.5. Les arcs pleins représentent les dépendances en valeur et les arcs en pointillé les dépendances en contrôle. Le point d'intérêt est situé en haut à droite du graphe 2.5 (*afficher_ entier(somme) ;*). Pour calculer la coupe de programme, nous

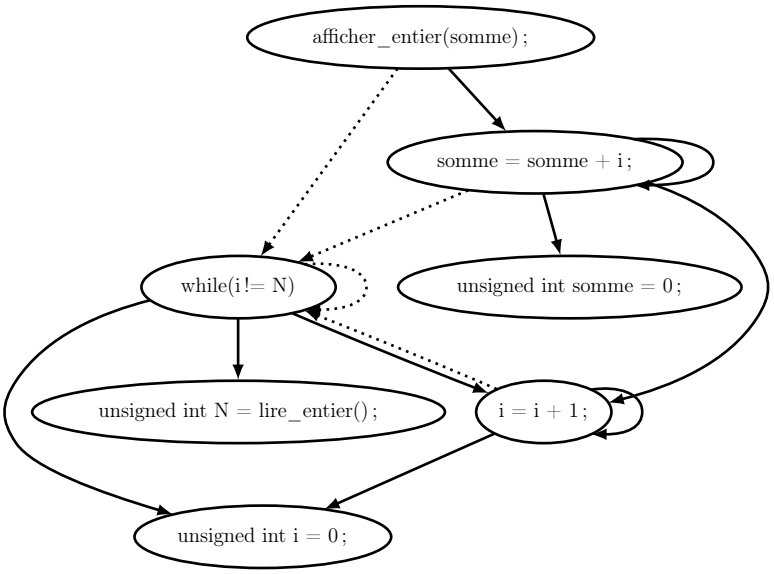


FIGURE 2.6 – Sous-graphe des noeuds accessibles

devons déterminé le sous-graphe contenant tous les noeuds accessibles à partir de ce dernier. Ceci peut-être fait à l'aide d'un algorithme de parcours de graphe. Une telle méthode consiste à explorer les sommets du graphe de proche en proche à partir du sommet initialement choisi. Il en existe plusieurs dont les plus couramment utilisés sont le parcours en profondeur, le parcours en largeur, les algorithmes gloutons et l'algorithme de Dijkstra [Dij59]. Le sous-graphe obtenu en résultat est représenté dans la figure 2.6.

Si maintenant on ne conserve du code source original que les instructions qui sont présentes dans ce sous-graphe et uniquement celles-ci, on obtient alors bien le programme réduit donné dans la figure 2.4 (colonne de droite). En conséquence, si le graphe de dépendance du programme permet de calculer une coupe de programme, il peut aussi permettre de déterminer l'ensemble des variables dont dépend un appel de fonction donné. En effet, il s'agit de l'ensemble des variables utilisées par les instructions présentes dans le sous-graphe des noeuds accessibles à partir de l'appel de fonction considéré.

2.3.3 Découverte des contraintes à vérifier

Maintenant que nous savons déterminer les ensembles de variables que l'on souhaiterait contrôler, il nous faut essayer de déterminer pour celles-ci des contraintes à vérifier. Il existe plusieurs types de propriétés qui peuvent affecter une variable (par exemple, un entier doit rester constant sur une certaine portion de code). Nous avons choisi de nous limiter dans ces travaux au domaine de variation des variables. Nous présentons dans les sections suivantes ce que nous voulons obtenir

exactement comme domaine de variation et comment ils peuvent être calculés par analyse statique du code source.

2.3.3.1 Les domaines de variation

Le domaine de variation d'une variable représente, à un point donné du programme, l'ensemble des valeurs qu'elle peut prendre. Nous avons vu dans la section 2.3.1 qu'une attaque contre les données de calcul peut forcer la valeur d'une variable hors de son domaine de variation. La connaissance de ces domaines de variation peut donc nous permettre de détecter des attaques de ce type. Cependant, nous avons également vu dans la section 2.3.1 que certaines de ces attaques pouvaient modifier illégalement une variable tout en respectant son domaine de variation mais que celles-ci pouvaient tout de même être détectées grâce à la connaissance des relations entre les variables. Nous allons voir dans cette section comment nous pouvons calculer statiquement, à partir du code source, des domaines de variation qui tiennent compte des relations entre les variables.

```
00. extern int a, b;
01.
02. void f(int);
03.
04. void g(){
05.
06.   if (b==0)
07.     a = 1;
08.   else
09.     if (b==1)
10.       a = 2;
11.     else
12.       return;
13.
14.
15.
16.   f(a);
17. }
```

```
00. extern int a, b;
01.
02. void f(int);
03.
04. void g(){
05.
06.   if (b==0)
07.     a = 1;
08.   else
09.     if (b==1)
10.       a = 2;
11.     else
12.       return;
13.
14.   assert((a==1 && b==0)
15.       || (a==2 && b==1))
16.   f(a);
17. }
```

FIGURE 2.7 – Exemple de code *C*

Pour cela, considérons maintenant comme exemple le code situé dans la figure 2.7 (colonne de gauche). Plus précisément, considérons l'appel à la fonction f de la ligne 16. Les variables dont dépend cet appel sont les entiers a et b. Notons que la variable a est une dépendance en valeur de l'appel à la fonction f, il s'agit en effet de son unique argument passé en paramètre. Notons également que la variable b est une dépendance en contrôle de l'appel à la fonction f, c'est sa valeur qui détermine si le chemin d'exécution va atteindre ou non cet appel. Si nous déterminons les domaines

de variation de ces deux variables, nous obtenons comme résultat que $a \in \{1, 2\}$ et que $b \in \{0, 1\}$.

Cependant, ceci n'est pas un résultat satisfaisant. En effet, nous avons perdu le lien de dépendance entre les variables a et b. Nous savons qu'il existe dans ce code plusieurs chemins possibles qui permettent d'atteindre l'appel à la fonction f considéré. Dans le cas général, chaque chemin possible peut présenter des propriétés différentes à l'exécution. Le schéma de la figure 2.8 représente le graphe des chemins d'exécution possibles pour le cas qui nous intéresse. On voit qu'il existe un chemin par lequel la variable a vaut 1, et que sur ce chemin la valeur de la variable b est nécessairement égale à 0.

De même, on voit qu'il existe un autre chemin par lequel la variable a vaut 2, et que sur ce chemin la valeur de la variable b est nécessairement égale à 1. Cette relation entre le chemin d'exécution et la valeur des variables a et b exprime la propriété de dépendance qu'il existe entre ces deux variables. C'est ce type de relation que l'on souhaite conserver afin d'exprimer des contraintes qui permettent de vérifier durant l'exécution du programme que les propriétés de dépendances que nous avons pu déterminer sont bien respectées.

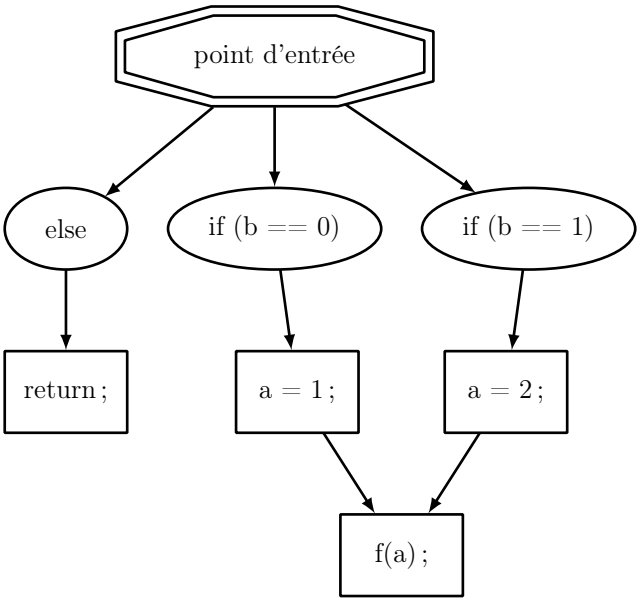


FIGURE 2.8 – Graphe des chemins d'exécution

En effet, considérons dans cet exemple la contrainte qui porte sur l'ensemble des variables dont dépend l'appel à la fonction f de la ligne 16. À partir du premier résultat, à savoir que $a \in \{1, 2\}$ et $b \in \{0, 1\}$, la contrainte que l'on peut déduire correspond au produit cartésien des domaines de variation de chacune des variables considérée séparément. Une telle contrainte permet de détecter une attaque qui

modifie la valeur d'une variable hors de son domaine de variation (comme l'attaque par corruption du mot de passe à la section 2.3.1.2), mais elle ne permet pas de détecter une attaque qui viole la relation de dépendance entre plusieurs variables (comme l'attaque par corruption de la variable de boucle à la section 2.3.1.2). La vérification des propriétés de dépendance entre les variables permet de détecter ce second type d'attaque.

Afin de permettre une meilleure détection, il nous faut déterminer des contraintes qui conservent les propriétés liant les variables entre elles. Pour cela, nous choisissons d'évaluer le domaine de variation de l'ensemble des variables non pas prises séparément, mais en tant que t-uplet. Si nous appliquons cela à l'exemple que nous utilisons dans cette section, nous obtenons comme résultat que $(a, b) \in \{(0, 1), (1, 2)\}$. Le lien de dépendance entre les variables a et b est bien conservé. L'assertion qui vérifie la valeur des variables a et b en tenant compte du lien de dépendance est présentée dans la figure 2.7 (colonne de droite, lignes 14 et 15). Comme nous l'avons illustré avec cet exemple, pour obtenir ce résultat pour un appel de fonction donné, nous devons d'abord déterminer l'ensemble des chemins qui permettent d'atteindre cet appel. Puis, idéalement, nous devons pour chacun de ces chemins pris séparément, déterminer le domaine de variation de chacune des variables dont dépend l'appel de fonction considéré.

2.3.3.2 L'interprétation abstraite

Une analyse statique exhaustive d'un programme doit permettre de prédire l'état du programme en tout point de son exécution et pour tous les scénarios d'exécution possibles. Cela revient à produire par le calcul l'ensemble des traces d'exécution possibles du programme. Les résultats de cette analyse permettent de vérifier que le programme respecte ou non une ou plusieurs contraintes données (par exemple, le programme ne doit jamais effectuer de division par zéro). Toutefois, réaliser une telle analyse pour un programme quelconque et dans un temps fini n'est pas possible dans le cas général.

En effet, il s'agit d'un problème indécidable (voir le théorème de Rice [Ric53] ainsi que le problème de l'arrêt [BM82]). Cependant, ce type d'analyse est pourtant nécessaire pour pouvoir connaître exactement le domaine de variation de chacune des variables manipulées par le programme jusqu'à un point donné de son exécution. Le calcul de ces domaines de variation souffre donc aussi de cette indécidabilité. Pour contourner ce problème, de nombreuses techniques ont été proposées parmi lesquelles l'interprétation abstraite [CC76, CC77a, CC77b, Cou02] est l'une des plus performantes et l'une des plus utilisées.

Cette approche repose sur une approximation de la sémantique du programme analysé. En effet, contrairement à la sémantique concrète qui décrit fidèlement l'exécution du programme, une sémantique abstraite ne raisonne plus en terme de valeurs mais en terme de propriétés (par exemple, suite à l'exécution d'une instruction,

plutôt que de déduire que $x = 1$, on se contente de savoir que $x \geq 0$). Ce qui permet de définir une sémantique abstraite, c'est l'ensemble des propriétés que celle-ci peut exhiber. C'est donc en limitant le nombre de propriétés que l'on s'assure de la calculabilité de cette sémantique. Cependant, cela implique également que les propriétés utilisées pour décrire les domaines de variation des variables sont des sur-approximations des propriétés concrètes. Les propriétés ainsi obtenues sont donc aussi dites abstraites.

Toute la difficulté de l'interprétation abstraite est de trouver le niveau d'abstraction qui permet à la fois de résoudre le problème de calculabilité de la sémantique du programme et de fournir en résultat des propriétés suffisamment précises pour que la vérification du respect des contraintes données ait un sens. En fait, il faut diminuer la précision de l'analyse pour rendre celle-ci réalisable mais pas trop au risque de ne pas être capable de répondre au problème posé. Une fois que ce niveau d'abstraction est défini, le calcul de la sémantique d'un programme revient de nouveau à calculer l'ensemble des traces d'exécution possibles du programme mais cette fois en raisonnant à l'aide des seules propriétés abstraites en lieu et place des propriétés réelles. Une fois l'analyse effectuée, les propriétés abstraites sont traduites en propriétés réelles (par exemple, $x \geq 0$ devient $x \in \{0, ..., 127\}$ si x est un entier codé sur 1 octet). Notons que cela implique que l'on doit être capable de traduire des propriétés concrètes en propriétés abstraites et réciproquement.

Un exemple simple d'abstraction pour les variables de type entier consiste à raisonner sur leur parité. Si nous limitons les opérations sur les entiers à petit nombre, par exemple à l'addition, à la soustraction, et à la multiplication, alors il est toujours possible de déterminer si le résultat est pair ou impair. Cette propriété abstraite, certes triviale et peu utile en pratique, permet d'illustrer sur un exemple l'interprétation abstraite.

En effet, durant le processus d'analyse, il est important de maintenir la correction de l'abstraction de chaque instruction du programme. Autrement dit, le processus d'abstraction doit préserver les propriétés de la sémantique concrète (par exemple, si une instruction change la parité d'une variable, cette propriété est conservée par la sémantique abstraite). Il est possible de garantir cette correction en utilisant une méthode formelle basée sur les fonctions monotones pour ensembles partiellement ordonnés et plus particulièrement les treillis [CC77a].

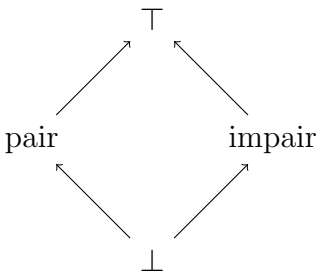
Plus précisément, définissons C comme étant l'ensemble des valeurs concrètes et $\mathcal{P}(C)$ comme l'ensemble des parties de C partiellement ordonnées par inclusion ($\subseteq$). Définissons également $\mathcal{A}$ comme l'ensemble des propriétés abstraites partiellement ordonnées par la relation $\sqsubseteq$. Soit γ la fonction qui représente chaque abstraction (un élément de $\mathcal{A}$) par le sous-ensemble de valeurs concrètes correspondantes. Cette fonction correspond au processus de concrétisation. Elle permet d'obtenir pour un ensemble de valeurs abstraites données un ensemble de valeurs concrètes. Soit α la fonction réciproque qui représente chaque sous-ensemble de valeurs concrètes par l'abstraction qui lui correspond le mieux. Cette fonction correspond au processus

d'abstraction. Elle permet d'obtenir pour un ensemble de valeurs concrètes donné un ensemble de valeurs abstraites. C'est sur ces deux fonctions que repose la correction de l'interprétation abstraite.

Cependant, un processus d'analyse statique reposant sur l'interprétation abstraite nécessite de procéder à des aller-retours entre les valeurs concrètes et les valeurs abstraites. Cela suppose que la composition de ces deux fonctions ($\gamma \circ \alpha$ et $\alpha \circ \gamma$) doit satisfaire un certain nombre de propriétés. En fait, il a été montré que pour garantir la correction de l'abstraction, lorsque que α et γ sont des fonctions respectivement monotones sur $\mathcal{P}(C)$ et $\mathcal{A}$, celles-ci doivent respecter les propriétés suivantes [CC77a] :

$$\begin{cases} \forall\, S \in \mathcal{P}(C) & S \subseteq \gamma \circ \alpha(S) \\ \forall\, a \in \mathcal{A} & \alpha \circ \gamma(a) \sqsubseteq a \end{cases}$$

La première propriété garantit que le processus d'abstraction est sûr dans le sens où il produit une sur-approximation. Plus précisément, elle garantit que l'abstraction d'un ensemble de valeurs concrètes donne un ensemble de valeurs abstraites qui correspond au mieux à l'ensemble de valeur concrètes contenant l'ensemble de valeurs concrètes de départ. Réciproquement, la seconde propriété garantit que le processus de concrétisation est sûr dans le sens où il produit une sous-approximation. Cette propriété est souvent limitée à une égalité stricte, à savoir $\alpha \circ \gamma(a) = a$, ce qui implique que toutes les propriétés abstraites sont exactes. Si nous reprenons l'exemple de la parité, nous pouvons définir $\mathcal{A}$ comme l'ensemble $\{\bot, \text{pair}, \text{impair}, \top\}$ associé à la relation d'ordre partiel suivante :



Notons que $\mathcal{A}$ est un un treillis que γ est alors définie par :

$$\begin{aligned} \gamma(\text{pair}) &= \{-\infty, \ldots, -4, -2, 0, 2, 4, \ldots, +\infty\} \\ \gamma(\text{impair}) &= \{-\infty, \ldots, -3, -1, 1, 3, \ldots, +\infty\} \\ \gamma(\top) &= [-\infty, +\infty] \\ \gamma(\bot) &= \emptyset \end{aligned}$$

Réciproquement, α est définie par :

$$\begin{aligned} \beta(2n) &= \text{pair} \\ \beta(2n+1) &= \text{impair} \\ \alpha(S) &= \bigcup \{\beta(v) \mid v \in S\} \end{aligned}$$

Appliquons maintenant ceci à un exemple simple. La figure 2.9 contient une suite d'instructions en langage C (ce sont les lignes numérotées). Le code de cette figure contient notamment une instruction de boucle (ligne 03) et une instruction de branchement conditionnel (ligne 04). L'objectif est de déterminer après analyse la parité des variables x et y. Grâce à l'interprétation abstraite, il est possible de déterminer que les variables x et y sont respectivement pair et impair à la fin de l'exécution du code. En effet, avant l'exécution de la boucle, les variables sont dans la situation inverse, à savoir respectivement impair et pair. Ni l'instruction exécutée systématiquement par la boucle (ligne 07) ni l'instruction exécutée dans la branche conditionnelle ne changent la parité de ces variables. Ceci arrive après l'exécution de la boucle, par les instructions ligne 09 et 10.

```
00. int x = 0;
 //@ ∃(k) ∈ ℕ, x = 2k
01. int y = x;
 //@ ∃(j, k) ∈ ℕ², x = 2j ∧ y = 2k
02. x = x + 1;
 //@ ∃(j, k) ∈ ℕ², x = 2j + 1 ∧ y = 2k
03. while (x > 8){
04.   if (f(x,y)){
 //@ ∃(j, k) ∈ ℕ², x = 2j + 1 ∧ y = 2k
05.     y = y + 2;
 //@ ∃(j, k) ∈ ℕ², x = 2j + 1 ∧ y = 2k
06.   }
 //@ ∃(j, k) ∈ ℕ², x = 2j + 1 ∧ y = 2k
07.   x = x + 2;
 //@ ∃(j, k) ∈ ℕ², x = 2j + 1 ∧ y = 2k
08. }
 //@ ∃(j, k) ∈ ℕ², x = 2j + 1 ∧ y = 2k
09. x = x + 1;
 //@ ∃(j, k) ∈ ℕ², x = 2j ∧ y = 2k
10. y = y + 1;
 //@ ∃(j, k) ∈ ℕ², x = 2j ∧ y = 2k + 1
```

FIGURE 2.9 – Exemple d'interprétation abstraite

L'utilisation de l'interprétation abstraite permet donc d'analyser un programme à partir d'une sur-approximation de sa sémantique concrète. Cette sur-approximation étant garantie, toutes les propriétés vérifiées par la sémantique abstraite le sont aussi par la sémantique concrète. En conséquence, ces propriétés calculées par interprétation abstraite peuvent alors être utilisées pour démontrer (ou non) qu'un programme respecte des propriétés concrètes données. L'exemple de la parité est trivial, en pratique on s'intéresse plutôt notamment à l'inférence de type ou encore à l'analyse d'intervalle.

2.4 Implémentation du système de détection

Pour tester notre approche pour la détection, nous avons implémenté la construction de notre modèle de détection dans un outil nommé *SIDAN* (pour *Software Instrumentation for the Detection of Attacks against Non-control-data* ou instrumentation logicielle pour la détection d'attaques contre les données de calcul). Pour le développement de cet outil, nous avons choisi de nous concentrer uniquement sur les programmes écrits en langage *C*.

En effet, ce type de programme est, en pratique, particulièrement susceptibles d'être la cible d'attaques contre les données de calcul. L'objectif de notre outil est de produire une automatisation complète, sans intervention humaine, du processus de génération des invariants ainsi que du processus d'instrumentation du code source. De plus, l'instrumentation ne doit pas nécessiter de modifier ensuite le processus de compilation du programme.

La génération des mécanismes de détection qui implémentent le modèle que nous proposons se déroule en trois étapes : le pré-traitement du code source, la génération des invariants et l'ajout d'assertions exécutables. Notre outil comprend tout d'abord un composant principal qui orchestre ces différentes étapes. Toutefois, pour la phase de génération des invariants, il fait appel à un composant dédié à cette tâche et nommé *Coninva* (pour *consistency invariants* ou invariants de cohérence). C'est ce composant qui procède à l'analyse statique du code source afin de générer les propriétés que l'on souhaite vérifier à l'exécution.

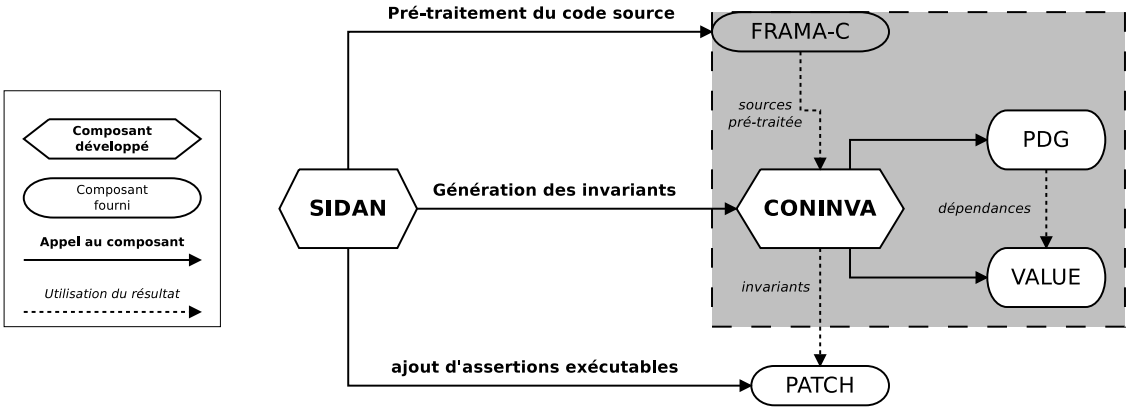


FIGURE 2.10 – Corruption de l'adresse de retour

Nous avons choisi de fonder l'implémentation de ce composant particulier sur un outil d'analyse statique existant et permettant de travailler sur les programmes écrits en langage *C*, *Frama-C* [FRAMA]. Ceci nous permet de ne pas avoir à réimplémenter nous même les techniques d'analyse statique sur lesquelles nous avons choisi de nous reposer pour construire notre modèle. Notons qu'il existe d'autres outils similaires

mais que nous avons toutefois opté pour ce dernier car son code source est librement accessible.

Dans la suite de cette section, nous commençons donc par présenter cet outil lorsque nous abordons la construction du modèle de détection. Comme nous l'avons vu dans la section 2.3.1.4, la génération des invariants se déroule en deux étapes : la découverte des ensembles de variables dont dépendent les appels de fonction et le calcul des contraintes qui s'appliquent à ces ensembles de variables. Chacune de ces étapes utilise les résultats d'un des composants de *Frama-C*. Nous les détaillons dans la suite de la section dédiée à la construction du modèle. Nous aborderons ensuite la phase d'instrumentation du programme qui comprend les étapes de pré-traitement du code source et d'ajouts d'assertions exécutables. Le figure 2.10 illustre par un schéma la composition de notre outil.

2.4.1 Construction du modèle

Nous allons maintenant détailler la phase de construction du modèle de comportement normal au sein de notre outil. Nous avons annoncé que cette phase repose sur le composant nommé *Coninva*. Celui-ci est implémenté sous forme d'un greffon *Frama-C* [FPLUG]. Le calcul d'invariants réalisé par ce composant repose sur l'exploitation de résultats fournis par d'autres greffons déjà présents dans l'outil d'analyse statique (voir la figure 2.10). Nous allons donc d'abord présenter cet outil ainsi que les greffons qui nous intéressent.

Pour un appel de fonction donné, la construction de l'invariant correspondant à notre modèle de comportement normal requiert deux étapes. Tout d'abord, le calcul de l'ensemble des variables dont dépend cet appel grâce aux résultats de la coupe de programme. Puis, deuxième étape, le calcul du domaine de variation de cet ensemble de variables en tant que t-uplet grâce aux résultats de l'interprétation abstraite. Ces deux étapes sont effectuées à chaque fois qu'un appel de fonction est rencontré dans le code source durant l'analyse syntaxique. Nous les détaillons maintenant dans la suite de cette section.

2.4.1.1 *Frama-C*

Frama-C signifie *Framework for Modular Analysis of C* ou plateforme modulaire pour l'analyse du *C*. Il s'agit d'un outil dédié à l'analyse de codes source écrits en langage *C* et qui est extensible via un système de greffons. Pour un programme donné, *Frama-C* permet de réaliser deux tâches principales : prouver que celui-ci respecte une spécification formelle et faciliter l'audit de son code source. Pour cela, *Frama-C* possède deux principaux greffons.

Tout d'abord, nous avons le greffon *Jessie* qui permet de prouver qu'un programme respecte bien la spécification annotée en commentaire. Ces annotations sont réalisées à l'aide d'un langage de spécification formel conçu pour le langage *C* et nommé *ACSL*. Le processus de vérification repose sur le calcul de la pré-condition

la plus faible, c'est-à-dire la pré-condition la moins restrictive qui garantisse la validité de la post-condition associée.

Puis, nous avons le greffon *Value analysis* qui lui permet de calculer le domaine de variation, en tout point atteignable d'un programme, de chaque variable accessible au point considéré. Notons que si un point particulier est situé dans du code considéré comme mort par l'analyse de valeur, alors nous ne pouvons pas obtenir de domaine de variation pour ce point. Notons que l'analyse effectuée par ce greffon est capable de parcourir plusieurs chemins en parallèle (ceci est configurable via un paramètre dédié appelé *slevel*). Elle n'est par contre pas capable de calculer le domaine de variation d'un t-uplet de variables. Le processus d'analyse de valeur repose sur des techniques d'interprétation abstraite.

Les autres greffons reposent tous sur les résultats obtenus par ces deux principaux greffons. C'est par exemple le cas du greffon *Program Dependence Graph* qui calcule pour chaque fonction du programme, grâce au résultat de l'analyse de valeur, un graphe de dépendance des instructions de la fonction. Le greffon *Slicing* utilise à son tour ce dernier pour effectuer des coupes de programmes.

2.4.1.2 Calcul des ensembles de variables

Nous avons vu à la section 2.3.2.3 que les variables dont dépend un appel de fonction donné correspondent à l'ensemble des variables utilisées par la coupe de programme ayant pour point d'intérêt cet appel de fonction. Cela comprend bien sûr les dépendances en valeur (les variables qui influencent la valeur des arguments passés en paramètre de l'appel) mais aussi les dépendances en contrôle (les variables qui influencent le chemin d'exécution utilisé pour atteindre l'appel). Nous avons également vu que pour répondre au problème d'accessibilité des variables dans le code source, nous ne voulons contrôler que les variables localement accessibles au moment de l'appel de fonction.

Une coupe de programme peut être calculée à l'aide du graphe de dépendance du programme (voir section 2.3.2.3). Dans *Frama-C* [FRAMA], le calcul d'une coupe de programme repose sur le calcul du graphe de dépendance de chacune des fonctions du programme. Le calcul de ce graphe de dépendance est implémenté dans le greffon *Program Dependance Graph*. Pour calculer l'ensemble de variables qui nous intéresse, nous utilisons donc d'abord ce greffon pour obtenir, au sein de la fonction contenant l'instruction qui nous intéresse, l'ensemble des instructions dont dépend cette dernière. Puis, nous extrayons de l'ensemble de ces instructions l'ensemble des variables utilisées par ces dernières. Nous obtenons ainsi, l'ensemble des variables localement accessibles dont dépend l'appel de fonction considéré.

2.4.1.3 Calcul des contraintes sur les ensembles de variables

Nous avons vu à la section 2.3.1 que les domaines de variation des variables pouvaient être des propriétés capables de détecter des tentatives d'intrusion. *Frama-*

C [FRAMA] propose le greffon *Value Analysis* pour calculer des domaines de variation. Nous nous reposons sur ce greffon pour calculer ces derniers. Toutefois, nous avons aussi vu à la section 2.3.3.1 que pour augmenter la capacité de détection, il fallait que les contraintes tiennent compte des relations entre les variables. Dans le cas d'un domaine de variation d'un ensemble de variables donné, cela implique que celui-ci soit calculé pour l'ensemble des variables en tant que t-uplet et non comme le produit cartésien de chacune des variables prises indépendamment. Or le comportement initial du greffon *Value Analysis* ne fournit pas en sortie ce type de contrainte.

En effet, *Frama-C* [FRAMA] n'explore par défaut qu'un chemin à la fois et pour chaque point du programme, ne conserve pas l'ensemble des états atteints par l'analyse mais seulement leur union. Le paramètre *slevel* permet d'augmenter le nombre de chemins parcourus en parallèle par l'analyse, repoussant ainsi le moment où il devient nécessaire de regrouper des états sous forme d'union. Cela se produit lorsque le nombre d'états atteints pour un point du programme est supérieur au nombre de chemins parcourus en parallèle. Si le nombre de chemins parcourus en parallèle est suffisant, l'analyse de valeur possède maintenant en interne toutes les informations nécessaires pour obtenir le type de contraintes que nous cherchons à déterminer.

Ces informations ne sont toutefois pas aisément accessibles car l'analyse ne les sauvegarde pas pour des raisons d'occupation mémoire. En interceptant le processus d'analyse, nous pouvons accéder à ces informations internes et sauvegarder nous-même les états qui nous intéressent. Comme nous n'avons besoin de faire cela que pour un sous-ensemble restreint des instructions du programme, à savoir les appels de fonctions dont les dépendances peuvent voir leur domaine de variation calculé, la surcharge en mémoire reste raisonnable. Au final, grâce à cette interception, nous pouvons calculer le domaine de variation d'un t-uplet de variables pour chaque chemin menant à l'instruction qui nous intéresse. Nous obtenons donc des contraintes similaires à celles données en exemple à la section 2.3.3.1.

Pour cela, notre greffon commence par enregistrer une fonction supplémentaire au sein de l'analyse de valeur afin de pouvoir obtenir ces informations. Cela se fait grâce à la fonction *Db.Value.Record_Value_Superposition_Callbacks.extend*. Cette fonction supplémentaire sauvegarde simplement dans une structure propre à notre greffon les états atteints par l'analyse de valeur lorsqu'il s'agit d'un appel de fonction. Une fois l'analyse de valeur terminée, notre greffon va utiliser *Frama-C* pour à nouveau parcourir l'arbre syntaxique abstrait. À chaque fois qu'un appel de fonction est rencontré, il utilise les résultats du greffon *Program Dependence Graph* pour obtenir l'ensemble des variables dont il dépend puis les résultats qu'il a sauvegardé durant l'analyse de valeur pour calculer le domaine de variation de cet ensemble, en tant que t-uplet.

Notons que, au sein d'un ensemble de variables, toutes ne pourront pas nécessairement faire l'objet d'une analyse de leur domaine de variation. Cela arrive lorsque

leur valeur dépend de données inconnues du point de vue du code source (par exemple, des entrées extérieures au programme ou le résultat d'appels à des bibliothèques externes). En pratique, la génération d'invariants effectuée par notre greffon s'applique au sous-ensemble des variables dont les domaines de variation ont pu être déterminés.

2.4.2 Instrumentation du programme

Afin que l'analyse de valeur se déroule correctement, y compris lorsque certaines fonctions appelées ne sont pas définies dans le code source, les spécifications de ces dernières doivent lui être fournies. Ces spécifications prennent la forme de petites fonctions *C* appelées fonctions bouchons (ou *stub*). Dans la suite de cette section, nous commençons par présenter la problématique de l'écriture de ces fonctions *stub* nécessaires à l'analyse de valeur.

Une fois que nous avons obtenu, pour l'ensemble du programme, les invariants statiquement calculables pour chacun des ensembles de variables dont dépendent les appels de fonction, il nous faut traduire ces invariants sous la forme d'assertions exécutables à ajouter dans le code source. Pour cela deux étapes sont nécessaires. La première étape est une phase de pré-traitement des fichiers source. Celle-ci a lieu avant la génération des invariants, en prévision de la deuxième étape. cette dernière correspond à la phase d'ajout des mécanismes de vérification des invariants implémentés sous forme d'assertions exécutables. Dans cette section, nous expliquons et détaillons ensuite ces deux étapes.

2.4.2.1 Écriture des fonctions *stub*

Il se peut que le code source que l'on cherche à analyser ne contienne pas la définition de toutes les fonctions qui y sont appelées. C'est notamment le cas lorsque le programme considéré fait appel à des bibliothèques de fonctions partagées. Dans la plupart des cas, cela ne va pas poser de problème à l'analyse de valeur. Prenons comme exemple la fonction *POSIX listen*. Le prototype de cette fonction est donné dans la figure 2.11.

```
int listen(int sockfd, int backlog);
```

FIGURE 2.11 – Prototype de la fonction *POSIX listen*

Sans définition plus précise de cette fonction, l'analyse de valeur va simplement supposer que le domaine de variation de la valeur retournée par un appel à la fonction *listen* correspond à l'ensemble de la plage de valeur du type *int*. Il s'agit là d'une sur-approximation. En effet, la norme *POSIX* spécifie que la fonction *listen* ne peut retourner que deux valeurs possibles, 0 ou -1. Néanmoins, grâce à la

sur-approximation de la valeur retournée, la méconnaissance de cette information n'empêche pas l'analyse de valeur de se dérouler correctement.

```
int getaddrinfo(const char *node, const char *service,
                const struct addrinfo *hints, struct addrinfo **res);
```

FIGURE 2.12 – Prototype de la fonction *POSIX getaddrinfo*

Ce n'est cependant pas toujours le cas. Prenons comme nouvel exemple la fonction *getaddrinfo*, une autre fonction *POSIX*. Le prototype de cette fonction est donné dans la figure 2.12. Cette fonction présente la particularité d'allouer une structure via un passage par adresse (il s'agit du dernier paramètre, nommé *res*). Or l'analyse de valeur ne peut pas supposer par défaut qu'à chaque passage par adresse d'un pointeur il s'agisse en fait d'une allocation. Toutefois, sans cette information, lorsque l'analyse de valeur rencontre une instruction qui accède à cette variable particulière, celle-ci considère qu'il s'agit d'un accès incorrect ayant pour conséquence la défaillance du programme. Lorsque cette situation se produit, l'analyse de valeur s'interrompt.

Pour répondre à ce problème, une première solution consiste à fournir une implémentation pour les fonctions concernées. En pratique, ceci n'est que rarement faisable. D'une part parce que le code à écrire est parfois complexe et d'autre part parce que cela pourrait augmenter grandement la taille du code à analyser et donc la durée d'analyse (par exemple, dans le cas des fonctions *POSIX*, il n'est pas envisageable de donner à analyser la totalité de la *glibc*).

```
int getaddrinfo(const char *node, const char *service,
                const struct addrinfo *hints, struct addrinfo **res){

        *res = malloc(sizeof(**res));

        return rand();
}
```

FIGURE 2.13 – *Stub* de la fonction *POSIX getaddrinfo*

Une autre solution consiste à fournir une courte fonction *C* qui présente les mêmes spécifications, vis-à-vis des données de sortie, que la véritable fonction. On désigne ces fonctions par le terme *stub*. Prenons toujours comme exemple la fonction *POSIX getaddrinfo*. On suppose ici que sa spécification vis-à-vis des données de sortie se résume à l'allocation du paramètre *res*. La fonction *stub* correspondant à cette spécification est présentée dans la figure 2.13. Si cette dernière est fournie en même temps que le code source à analyser, bien que son comportement diffère de la véritable fonction *getaddrinfo*, alors l'analyse de valeur ne s'interrompra plus et pourra aller jusqu'à son terme.

```
int listen(int sockfd, int backlog){
        return alea_bin_err(-1,0,-1,
                            ((int[]){EADDRINUSE,EBADF,ENOTSOCK,EOPNOTSUPP}));
}
```

FIGURE 2.14 – *Stub* de la fonction *POSIX listen*

Notons que si l'écriture d'un *stub* est indispensable dans le cas de certaines fonctions, le faire systématiquement pour chaque fonction uniquement prototypée présente toutefois un intérêt. En effet, reprenons cette fois l'exemple de la fonction *POSIX listen*. La sur-approximation consistant à considérer que le domaine de variation de la valeur retournée correspondait en fait à l'ensemble de la plage de valeur du type *int* est suffisante pour conduire l'analyse de valeur jusqu'à son terme. Cependant, un tel domaine de variation ne nous permet pas d'obtenir un invariant. Puisque la spécification de la fonction *listen* nous dit que celle-ci ne peut retourner que deux valeurs entières possibles, 0 ou -1, l'écriture d'un *stub* peut nous permettre d'améliorer les résultats de l'analyse de valeur.

Il est possible d'aller plus loin dans le respect de la spécification. En effet, dans le cas de la fonction *listen*, la norme *POSIX* nous dit que cette dernière retourne 0 en cas de succès et -1 en cas d'erreur auquel cas la variable *errno* prend l'une des valeurs suivantes : *EADDRINUSE*, *EBADF*, *ENOTSOCK* ou *EOPNOTSUPP*. La fonction *stub* correspondant à cette spécification est présentée dans la figure 2.14. On y fait usage de la macro *alea_bin_err*. Il s'agit d'une macro que nous avons écrit pour simplifier l'écriture des fonctions *stub*. Cette dernière retourne aléatoirement un de ses deux premiers paramètres et lorsque la valeur retournée est égale au troisième paramètre, alors la variable *errno* reçoit aléatoirement l'une des valeurs de la liste fournie en quatrième paramètre.

Avec plusieurs macros de ce type, correspondant aux différents cas possibles, nous avons écrit en tout 160 fonctions *stub*. Nous avons également fourni 38 véritables implémentations. Ces dernières correspondent à des fonctions très simples à implémenter mais qui permettent néanmoins d'améliorer de manière significative les résultats de l'analyse de valeur. C'est notamment le cas de certaines fonctions qui manipulent des entiers (par exemple, *ntohs*) ou des chaînes de caractères (par exemple, *strcpy*). Au final, implémentation ou *stub*, pour l'ensemble des programmes que nous avons instrumentés, l'analyse de valeur a toujours obtenu une définition des fonctions rencontrées.

Notons qu'une fonction se présente comme un cas particulier, la fonction *malloc*. Cette fonction est chargée de réaliser les allocations mémoire. Le programme analysé peut y faire appel, mais les fonctions *stub* également. C'est par exemple le cas de notre fonction *stub* correspondant à la fonction *POSIX getaddrinfo* (voir figure 2.13). Par défaut, *Frama-C* fournit quatre implémentations au choix pour cette fonction :

– *FRAMA_C_MALLOC_INDIVIDUAL*,

– *FRAMA_C_MALLOC_CHUNKS*,
– *FRAMA_C_MALLOC_POSITION*,
– *FRAMA_C_MALLOC_HEAP*.

Les deux premières nécessitent l'assurance que le nombre d'appels à la fonction *malloc* est fini. Ce n'est par exemple pas le cas si un appel à cette fonction est placée dans une boucle et que le critère d'arrêt n'est pas borné. Dans le cas général, ce n'est pas quelque chose que nous pouvons assurer. Nous avons donc choisis de ne pas les utiliser pour notre outil. La troisième implémentation, quant à elle, fournit également une fonction *stub* pour la fonction *free*. Cette dernière utilise une heuristique pour effectuer des vérifications sur la libération de l'espace mémoire alloué. Ceci n'est pas quelque chose dont nous avons besoin (nous ne cherchons pas ici à détecter des vulnérabilités mais à calculer des invariants). Nous avons donc tout d'abord utilisé pour notre outil la quatrième implémentation : *FRAMA_C_MALLOC_HEAP*. Le code de la fonction *malloc* correspondante est donné dans la figure 2.15.

```
#define MEMORY_SIZE 100000000
char MEMORY[MEMORY_SIZE];

void *malloc(size_t size){
  static int next_free = 0;
  next_free += size;
  if(next_free>=MEMORY_SIZE)
    return NULL;
  return (MEMORY+(next_free-size));
}
```

FIGURE 2.15 – *Stub* original de la fonction *malloc*

Notons tout de même que cette fonction nous pose deux problèmes. Tout d'abord, l'allocation mémoire est faîte sur un tableau de caractères déclaré comme une variable globale. Or, d'après la norme *C99*, les variables globales doivent être initialisées à la valeur 0, y compris les tableaux. Cela a pour conséquence que l'analyse de valeur peut nous indiquer que des variables sont initialisées à 0 alors qu'elles devraient être à l'état *UNINITIALIZED*. Ceci peut potentiellement conduire à la génération d'invariants erronés. Ensuite cette fonction ne peut retourner la valeur *NULL* que si le total des allocations mémoire dépasse la valeur *MEMORY_SIZE*. Cette fonction ne prévoit donc pas d'autres sources d'erreurs qu'un manque de mémoire. Cela a pour conséquence que les blocs de code dédiés au traitement de cette erreur ne sont pas traités par l'analyse de valeur. Cependant, bien que ces blocs de code soient problématiques pour nos campagnes de tests (car difficilement atteignables), nous choisissons quand même de les instrumenter (voir section 4.1.1.3). Au final, l'implémentation de la fonction *malloc* que nous utilisons est une modification de la version originale donnée dans la figure 2.15. En plus du traitement original, celle-ci introduit systématiquement la valeur *NULL* comme une valeur de retour possible et initialise

chacun des éléments de la mémoire allouée avec l'ensemble de la plage de valeur d'un caractère.

2.4.2.2 Pré-traitement des fichiers sources

Cette étape a pour but de préparer le code source du programme afin de le rendre apte à être modifié. Il y a plusieurs raisons à cela. D'abord parce que *Frama-C*, pour conduire son analyse, modifie légèrement le programme. En effet, celui-ci peut parfois ajouter des variables supplémentaires. Ces variables ainsi ajoutées peuvent alors faire partie des dépendances d'un ou plusieurs appels de fonction et peuvent donc apparaître dans les invariants. C'est notamment le cas lorsque la valeur retournée par un appel de fonction est utilisée directement, sans passer par une variable intermédiaire. La figure 2.16 illustre ce cas. Dans cet exemple, la variable supplémentaire *tmp* est déclarée à la ligne 02 (colonne de droite). Cette dernière influence l'appel de fonction $printf$ à la ligne 05 (le code original est situé dans la colonne de gauche).

Si un invariant tenant compte du domaine de variation de cette variable supplémentaire peut être calculé, alors cet invariant ne peut être vérifié par une assertion exécutable ajoutée au code original. Nous pourrions supprimer dans les invariants ces variables supplémentaires. Cependant, les utiliser nous permettrait d'augmenter le nombre d'invariants ainsi que la précision de ces derniers. Nous choisissons donc d'inclure l'ajout de ces variables supplémentaires dans la phase de pré-traitement du code source. Pour cela, la première étape consiste alors à appliquer le pré-traitement de *Frama-C* à chacun des fichiers source.

```
00. void f(const char *s){
01.
02.
03.
04.
05.   printf("taille:%i",strlen(s));
06.
07.   return;
08. }
```

```
00. void f(const char *s){
01.
02.   size_t tmp ;
03.   tmp = strlen(s);
04.
05.   printf("taille:%i", tmp);
06.
07.   return;
08. }
```

FIGURE 2.16 – Exemple de variable ajoutée

Ensuite, il existe un autre cas où un invariant peut porter sur une variable qui n'est pas déclarée. Celui-ci est illustré par l'exemple de la figure 2.17. L'appel à la fonction f (ligne 06) étant placé dans une boucle, celui-ci peut dépendre d'instructions qui ne sont exécutées qu'après lui. Ces instructions peuvent utiliser des variables qui n'étaient pas jusqu'alors utilisées par les instructions précédant l'appel de fonction considéré. Ces variables font toutefois partie des dépendances de cet

appel de fonction et peuvent donc apparaître dans l'invariant correspondant. Si ces variables ne sont déclarées qu'après l'appel de fonction considéré (par exemple, juste avant leur utilisation), alors l'invariant ne pourra pas être vérifié par une assertion exécutable ajoutée au code source car certaines des variables sur lesquelles il porte ne sont pas encore déclarées. Un tel placement des déclarations est autorisé selon la norme C99 [C99].

```
00. int a = 1;
01.
02. while(a){
03.
04.
05.
06.   f();
07.   int b;
08.   b = g();
09.   a = h(b);
10. }
```

```
00. int a = 1;
01.
02. while(a){
03.
04.   int b;
05.
06.   f();
07.
08.   b = g();
09.   a = h(b);
10. }
```

FIGURE 2.17 – Exemple de dépendance dans une boucle

Dans l'exemple de la figure 2.17, c'est le cas de la variable b. Celle-ci n'est déclarée qu'après l'appel à la fonction f mais elle influence l'évaluation de la condition de boucle ligne 02 (la valeur de la variable a dépend d'elle). Elle fait donc partie des dépendances de l'appel à la fonction f ligne 06 puisqu'il faut rentrer dans la boucle pour accéder à cet appel. Pour résoudre ce problème, là encore nous pourrions détecter cela et supprimer ce type de variable des invariants.

À nouveau, nous préférons les garder afin d'augmenter le nombre d'invariants ainsi que la précision de ces derniers. Pour cela, nous choisissons de déplacer toutes les déclarations de variables au début du bloc correspondant. En fait, nous ne cherchons pas à détecter quelles déclarations de variable ont besoin d'être déplacées, nous les déplaçons toutes. Ceci est réalisé à l'aide de *Frama-C* en même temps que nous appliquons son pré-traitement. Dans l'exemple de la figure 2.17, la colonne de droite illustre comment nous déplaçons les déclarations de variable.

Ces deux premiers pré-traitements, à savoir l'ajout de variables supplémentaires et le déplacement des déclarations de variables, modifient la sémantique opérationnelle du programme. En effet, dans le premier cas, pour chaque variable supplémentaire ajoutée au code source, une opération d'écriture supplémentaire est également ajoutée et l'espace mémoire à réserver pour les variables locales est nécessairement augmenté. Dans le second cas, si des déclarations de variables sont déplacées, alors les instructions utilisées pour réserver l'espace mémoire de ces variables le sont potentiellement aussi.

Toutefois, même si nous n'avons pas étudié formellement la question, on peut

raisonnablement supposer que ces modifications n'impliquent pas que la sémantique dénotationelle du programme ne soit modifiée. Effectivement, en situation de fonctionnement normal, les résultats produits par le programme ne devraient pas être affectés par ces modifications. Notons cependant qu'à moins de travailler avec une chaîne de compilation prouvée [CCERT], nous n'avons de toute façon aucune assurance que la sémantique du code source sera respectée. Par exemple, le déplacement des déclarations de variables fait partie des opérations d'optimisation que réalise déjà le compilateur *GCC* avec lequel nous travaillons.

Notons qu'en situation de fonctionnement normal, le même raisonnement peut être appliqué aux assertions exécutables que nous ajoutons au code source. En effet, sauf si nous sommes en présence d'attaques, la vérification des invariants que nous avons calculés ne modifie pas l'espace mémoire du processus (seules des opérations de lecture ont accès aux données utilisées pour stocker la valeur des variables présentes dans le code source).

Si par contre une attaque est détectée, le comportement du programme est effectivement modifié (une alerte est levée puis le processus continue son exécution). Cependant, il s'agit là de l'objectif même d'un système de détection d'intrusion embarqué dans le programme. Au final, l'important est de pouvoir considérer que pré-traitement et instrumentation ne modifient pas le comportement du programme tant que celui-ci est conforme au modèle de référence préalablement calculé par l'analyse statique.

Pour finir, les fichiers source sont pré-traités car il nous faut ajouter dès maintenant les entêtes nécessaires aux assertions exécutables que nous ajoutons au code. En effet, les assertions exécutables peuvent faire appel à des fonctions de la bibliothèque standard du langage *C*. Or le pré-traitement de *Frama-C* repose en partie sur le préprocesseur du compilateur *C* et celui-ci ajoute avant le code une copie des entêtes. Il n'est alors plus possible d'ajouter les entêtes nécessaires sans risquer de dupliquer des déclarations de fonction (cela arrive lorsqu'une des entêtes nécessaires était déjà présente dans le fichier original) ce qui provoque une erreur à la compilation. Plutôt que de chercher, après le pré-traitement de *Frama-C*, à ajouter uniquement les entêtes manquantes, toutes les entêtes nécessaires sont ajoutées aux fichiers source avant de faire appel à *Frama-C*.

2.4.2.3 Ajout des assertions exécutables

Une fois chaque fichier source pré-traité, une assertion exécutable est ajoutée avant chaque appel de fonction si un invariant a pu être calculé sur une ou plusieurs des variables appartenant à l'ensemble de ses dépendances. Durant le parcours de l'arbre syntaxique abstrait créé par *Frama-C*, celui-ci est capable de nous donner le fichier d'origine ainsi que la ligne d'une instruction donnée. Nous utilisons donc ces informations ainsi que les invariants obtenus pour générer des correctifs qui ajoutent à chaque fichier source les éventuelles assertions exécutables.

```
void f(int a, int b){

  int c;
  int d;

  if (a==0)
    c = 1;
  else
    if (a==1)
      c = 2;
    else
      return -1;

  if (b==1)
    d = 0;
  else
    if (b==2)
      d = 2;
    else
      return -1;

  coninva_assert(0x00000010,
       (d==0 && b==1 && c==1 && a==0)
    || (d==0 && b==1 && c==2 && a==1)
    || (d==2 && b==2 && c==2 && a==1)
    || (d==2 && b==2 && c==1 && a==0)
  );

  g(a,c,b,d);

  return 0;
}
```

FIGURE 2.18 – Exemple de vérification d'invariant

Ces assertions exécutables possèdent un identifiant unique qui correspond au premier paramètre de la fonction appelée pour lever une alerte. Il s'agit du numéro de l'instruction dans l'arbre syntaxique abstrait utilisé par *Frama-C*. Sur le même modèle que la macro *assert*, standard au langage *C*, le code qui vérifie l'invariant est inséré directement dans le flot de code de la fonction appelante. La figure 2.18 illustre cela sur un exemple. L'assertion exécutable obtenue correspond ici au contrôle de l'appel à la fonction *g*.

La fonction appelée par l'assertion exécutable est en fait une macro *C* qui appelle la véritable fonction chargée de lever une alerte. Cela permet, lorsque le cas se produit, de fournir à cette fonction un certain nombre d'informations telles que l'invariant non vérifié à l'origine de l'alerte ou bien encore la localisation de l'assertion qui l'a provoquée. La figure 2.19 présente la macro et la fonction utilisées par défaut dans *SIDAN* pour lever une alerte. On peut voir que la fonction utilisée par l'assertion exécutable n'est appelée que lorsque l'invariant n'est pas vérifié. Si aucun incident ne se produit durant l'exécution du programme, à aucun moment ce code n'est amené à être exécuté.

2.5 Résumé et discussion

Nous avons présenté dans ce chapitre une approche pour la détection d'intrusion hôte. Plus précisément, nous avons proposé un système de détection d'intrusion de niveau applicatif qui cible un type d'attaques bien particulier, les attaques contre les données de calcul. Il s'agit des attaques réalisées au travers de chemins d'exécution

```
#define coninva_assert(id, expr) \
  (if (expr) \
     __coninva_assert(id, #id": "#expr, __FILE__, __LINE__))

void __coninva_assert(int id, int char *assert, char *file, int line){

  fprintf(stderr, "[coninva] coninva_assert_failed %s:%i -> %s\n",
                  file, line, assert);
  fflush(stderr);
}
```

FIGURE 2.19 – Code par défaut de levée d'alerte

valides vis-à-vis de la spécification des programmes. Pour arriver à ce résultat, un utilisateur malveillant va chercher à corrompre les données utilisées pour contenir la valeur des variables issues du code source. C'est ce que nous désignons comme des données de calcul.

Le modèle sur lequel repose notre mécanisme de détection est un modèle de comportement normal construit par analyse statique du code source des applications. Plus précisément, notre modèle est orienté autour des variables. En effet, l'analyse que nous effectuons a pour but de calculer des invariants sur les variables que pourrait chercher à corrompre un utilisant malveillant (via les données de calcul utilisées pour contenir leur valeur).

La détection s'opère en ligne, durant l'exécution des programmes surveillés. Pour cela, des assertions exécutables sont dérivées du modèle de détection préalablement construit avant d'être insérées dans le code source des programmes. Ce sont donc les programmes eux-même qui embarquent individuellement les mécanismes de vérification et qui lèvent des alertes lorsqu'une déviation comportementale est observée. Les programmes ainsi obtenus sont dits durcis vis-à-vis des attaques contre les données de calcul.

Dans ce chapitre, nous avons également présenté une implémentation de notre approche. Nous avons entre autres expliqué comment répondre aux problèmes posés par la réalisation de cette implémentation. Celle-ci repose sur *Frama-C* [FRAMA], un outil d'analyse statique pour les programmes écrits en langage *C*. En effet, il s'agit là de programmes qui sont en pratique susceptibles d'être la cible d'attaques du type qui nous intéresse.

Chapitre 3

Modèle de simulation d'attaque

Le mécanisme de détection d'intrusion que nous avons présenté dans le chapitre précédent cible spécifiquement les attaques contre les données de calcul. Celui-ci repose sur un modèle de détection comportemental afin de détecter des attaques encore inconnues au moment de la modélisation. Le comportement de l'application est modélisé à l'aide d'invariants portant sur les variables utilisées dans le code source pour stocker les informations dont dépendent les appels de fonction. Le mécanisme que nous proposons pour construire un tel modèle de comportement repose sur des techniques d'analyse statique qui nous assurent que les invariants calculés sont des sur-approximations (c'est-à-dire que ces derniers sont toujours vérifiés en situation de fonctionnement normal).

Cela implique que le modèle de détection ainsi obtenu est un modèle complet (voir section 1.1.2) et donc que le taux de faux positifs est toujours nul quelle que soit l'application à laquelle est appliquée notre approche pour la détection. Toutefois, notre modèle n'offre aucune garantie quant à sa correction (voir section 1.1.2) et peut donc présenter des faux négatifs. Afin d'évaluer plus en détails la capacité de détection du système que nous proposons, nous voulons donc également connaître le taux de faux négatifs.

Cependant, afin de pouvoir calculer pour un programme donné le taux de faux négatifs d'une approche de type comportemental, il nous faudrait connaître toutes les vulnérabilités présentes dans le code source de ce programme ainsi que toutes les attaques réalisables pour chacune d'entre elles. Ceci n'est pas possible, ni automatiquement ni manuellement. C'est pourquoi nous proposons également dans ces travaux de thèse une approche permettant d'estimer la capacité de détection des mécanismes hôtes reposant sur un modèle de comportement normal et ce vis-à-vis des d'attaques contre les données de calcul.

Pour cela, notre approche pour l'évaluation repose sur la simulation des erreurs

logicielles engendrées par ce type d'attaques. Nous proposons pour implémenter notre approche d'utiliser un mécanisme d'injection de fautes. La simulation de l'attaque est donc réalisée sans exploiter de vulnérabilités, mais en allant directement modifier les données de calcul que pourrait tenter de corrompre un attaquant. En simulant ainsi un grand nombre d'attaques, nous pouvons alors, pour un programme donné, donner une estimation du taux de couverture de notre mécanisme de détection des intrusions.

Dans ce chapitre, nous commençons par expliquer comment nous proposons de simuler les erreurs logicielles engendrées par les attaques contre les données de calcul. Notre approche pour la simulation d'attaques reposant sur la capacité à injecter des fautes durant l'exécution du programme, nous présentons d'abord le modèle de faute que nous proposons puis la manière dont nous avons choisi de reposer sur ce modèle pour évaluer le taux de couverture de notre mécanisme de détection d'intrusion. Nous présentons ensuite comment nous avons implémenté la simulation d'attaques dans le cadre des programme écrits en langage *C*. Le système d'injection que nous proposons reposant sur l'instrumentation du programme, nous présentons d'abord comment nous construisons les mécanismes d'injection puis comment ces derniers doivent être utilisés durant la procédure de tests. Enfin, nous résumons les contributions présentées dans ce chapitre et nous discutons de la pertinence de nos simulations vis-à-vis des attaques réelles mais aussi vis-à-vis de l'évaluation des mécanismes de détection d'intrusion.

3.1 Simulation d'erreurs sur les données de calcul

Bien qu'il soit possible de tester un système de détection d'intrusion de type comportemental sur des cas réels d'attaques contre les données de calcul [CXS+05], il n'est pas possible de connaître exactement son taux de couverture et ce même pour un programme précis. En effet, il n'est bien sûr pas possible de connaître automatiquement toutes les attaques de ce type dont pourrait être la cible un programme donné et une méthode qui ne reposerait que sur des cas réels ne couvrirait qu'un ensemble restreint d'attaques contre les données de calcul, celles qui sont connues à un instant donné.

Cependant, l'un des principaux avantages des modèles comportementaux est justement de pouvoir détecter des attaques encore inconnues au moment de la modélisation. C'est pourquoi nous proposons de simuler le résultat d'attaques ciblant les données de calcul dans un programme donné et ce sans connaissance *a priori* des vulnérabilités qui le touchent. Nous pourrons alors grâce à cette approche produire une estimation de son taux de couverture.

Nous avons défini dans le chapitre précédent l'état interne d'un programme comme l'ensemble des éléments qui constituent son espace mémoire. Nous avons également énoncé dans ce chapitre qu'une attaque rend nécessairement incohérent

cet état interne vis-à-vis de la spécification du programme. Lors d'un cas réel d'attaque, c'est grâce à l'exploitation d'une vulnérabilité présente dans le programme que l'attaquant est capable de perturber cet état interne. Afin de simuler le résultat d'une attaque (réussie ou non, c'est-à-dire permettant ou non une intrusion), nous proposons de perturber artificiellement l'état interne du programme en allant directement modifier certains de ses éléments. Les mécanismes qui permettent ce type de corruption reposent sur des techniques d'injection de fautes en mémoire (voir la section 1.4.2.3). Nous allons donc avoir recours à un mécanisme de ce type pour procéder à la simulation d'attaques contre les données de calcul.

Le modèle de faute qui doit être utilisé avec ce mécanisme d'injection en mémoire a pour objectif de reproduire les erreurs engendrées par des attaques contre les données de calcul. Dans la suite de cette section, nous allons donc commencer par présenter ce modèle. Pour cela, nous parlerons d'abord des caractéristiques de ce type d'attaques. Le mécanisme d'injection de fautes que nous proposerons devra donc être capable de simuler ces caractéristiques. Nous parlerons aussi de l'impact sur le programme de ces simulations d'attaques : combien de temps l'état interne du programme est-il effectivement invalide durant son exécution et dans quel cas cela engendre-t-il une déviation comportementale ? Nous discuterons ensuite des cas où l'injection va pouvoir être détectée par notre mécanisme de détection. Puis, nous expliquerons comment nous proposons de construire les mécanismes d'injection qui implémentent le modèle de faute que nous proposons.

Pour finir, nous présentons dans cette section une méthode pour évaluer le taux de faux négatifs de notre approche et qui nous assure que le résultat obtenu est une sur-approximation du taux réel de faux négatifs. Cette méthode est conçue pour être utilisée avec notre modèle de faute lors d'une campagne de tests. La sur-approximation est obtenue en nous assurant que le système de détection d'intrusion testé opère dans la situation la moins favorable possible pour lui. Pour respecter cette contrainte, nous justifierons les valeurs utilisées pour corrompre une donnée particulière mais aussi le nombre de cibles et le nombre d'injections à effectuer lors de chaque test. Nous discuterons enfin des avantages et des limites de notre modèle de faute dans cette situation.

3.1.1 Modèle de fautes

Pour simuler le résultat d'attaques contre les données de calcul, nous proposons d'avoir recours à un mécanisme d'injection de fautes. Cela implique donc que nous devons disposer d'une abstraction modélisant l'impact de ce type d'attaques sur les programmes. C'est ce que l'on désigne par modèle de faute et nous allons maintenant présenter dans la suite de cette section celui que nous proposons pour simuler directement au niveau de l'espace mémoire des processus le résultat d'attaques contre les données de calcul. Ce choix implique donc l'utilisation d'un mécanisme d'injection en mémoire.

Rappelons que si nous cherchons à simuler le résultat d'attaques contre les données de calcul, c'est dans le but d'évaluer le taux de faux négatifs de notre modèle de détection. Ce modèle ayant pour objectif de détecter des attaques encore inconnues au moment de la modélisation, cette évaluation ne peut être qu'une estimation. Nous faisons le choix d'obtenir une sur-approximation du taux de faux négatifs. Ainsi, nous pourrons comparer le mécanisme que nous proposons avec d'autres systèmes de détection d'intrusion en plaçant notre approche dans la situation la moins favorable.

Cette décision va notamment influencer un certain nombre de choix dans la conception du modèle mais aussi dans la manière de le mettre en œuvre lors de l'évaluation. En effet, à chaque fois que des choix doivent être faits, nous choisissons toujours celui qui place notre mécanisme de détection d'intrusion dans la situation la moins favorable possible afin de s'assurer que l'estimation ainsi obtenue est bien une sur-approximation du taux réel de faux négatifs.

Pour présenter notre modèle de faute, nous allons d'abord détailler les caractéristiques du type d'attaques que nous voulons simuler et qui devront être reproductibles par un mécanisme d'injection en mémoire. Nous abordons ensuite la question de l'impact sur le programme en cours de test des fautes simulées selon un modèle qui reproduit ces caractéristiques. Pour cela nous abordons les problématique d'annulation et de masquage d'erreurs. Puis, nous expliquons comment construire notre modèle de faute pour un programme donné avant de discuter des avantages et des limitations du modèle ainsi obtenu. Enfin, nous présentons comment il est possible de reposer sur ce modèle de faute pour procéder à l'évaluation de notre système de détection d'intrusion vis-à-vis des attaques contre les données de calcul et comment nous avons implémenté les mécanismes d'injection dans le cas des programmes écrits en langage *C*.

3.1.1.1 Caractéristiques à simuler

Dans la section 2.3.1.1, nous avons illustré le déroulement d'une attaque contre les données de calcul à l'aide d'un exemple reposant sur l'exploitation d'une vulnérabilité présente dans une ancienne version du serveur *SSH* libre *OpenSSH* [Zal01]. Correctement exploitée, cette vulnérabilité peut permettre à un utilisateur malveillant de modifier n'importe quel élément de l'espace mémoire du processus en cours d'exécution. De plus, l'élément ciblé par l'attaquant peut être modifié directement, c'est-à-dire sans avoir besoin de modifier préalablement d'autres éléments mémoire pour atteindre ce dernier.

Les attaques qui exploitent ce genre de vulnérabilités ont donc la possibilité de modifier sans restriction l'ensemble des données du processus. Si l'objectif est de corrompre l'état interne du processus pour en détourner l'exécution, il s'agit là du plus haut niveau de sévérité pour une vulnérabilité. Toutes les vulnérabilités n'ont pas nécessairement ce niveau de sévérité, néanmoins grâce à cet exemple, nous savons

que la pire situation peut réellement se produire en conditions réelles. Puisque nous voulons nous placer dans la situation la moins favorable lorsque nous allons simuler le résultat d'une attaque contre les données de calcul, nous allons considérer qu'à tout moment de l'exécution du programme une vulnérabilité ayant ce niveau de sévérité existe. Autrement dit, nous allons simuler uniquement des attaques qui pour être réalisées nécessitent à un moment donné d'avoir accès en écriture à l'ensemble de l'espace mémoire du processus en cours d'exécution.

Cependant, rappelons que notre objectif n'est pas de simuler n'importe quel type d'attaques mais seulement une famille bien précise d'attaques, celles qui pour être réalisées nécessitent de corrompre des données de calcul. C'est-à-dire que même si nous allons simuler le résultat d'une attaque reposant sur l'exploitation d'une vulnérabilité donnant un accès complet et direct à l'ensemble de l'espace mémoire du processus, tous les éléments de cet espace mémoire ne constituent pas des cibles potentielles.

En effet, pour simuler le résultat d'une attaque contre les données de calcul, nous ne voulons pas modifier n'importe quel élément de l'espace mémoire du processus, mais uniquement les éléments utilisés pour contenir la valeur des variables dont dépendent les appels système. Ceci va ainsi nous permettre de conduire le programme en cours d'exécution dans un état interne invalide mais néanmoins représentatif du résultat d'une attaque contre les données de calcul.

Toutefois, maintenant que nous avons déterminé les cibles potentielles qui sont caractéristiques d'une attaque contre les données de calcul, une question se pose : doit-on ou non lors de l'injection tenir compte des valeurs originellement contenues dans les éléments ciblés. En effet, on peut supposer qu'il existe des attaques, qui, pour être réalisées, nécessitent que les données erronées envoyées par l'utilisateur malveillant soient calculées en fonction de ces informations originales. Ce n'est par exemple pas le cas des exemples d'attaques que nous avons présentés dans la section 2.3.1.1. Cependant, il s'agissait d'attaques qui pour être réalisées ne nécessitaient qu'une seule exploitation de la vulnérabilité présente dans le programme.

Or, une attaque qui aurait pour objectif de corrompre une donnée particulière avec une nouvelle valeur qui dépendrait de la valeur contenue à l'origine dans l'élément ciblé ne pourrait se faire que via deux exploitations de la vulnérabilité : une première exploitation est nécessaire pour récupérer la valeur originale (violation de la confidentialité) puis une seconde exploitation permet de corrompre la cible avec la nouvelle valeur (violation de l'intégrité).

Nous allons voir dans la section 3.1.2 que pour se placer dans la situation la moins favorable nous ne devons simuler que des attaques qui pour être réalisées ne nécessitent qu'une seule exploitation de la vulnérabilité. Cela implique donc, toujours dans un souci de se respecter ce critère, de supposer que pour être réalisée une attaque n'a pas besoin de connaître tout ou partie de l'état interne original du programme. Au final, notre modèle ne repose donc pas sur la valeur originalement contenue par les cibles potentielles.

3.1.1.2 Impact des fautes sur le programme

Dans le cas du mécanisme de détection que nous avons présenté au chapitre 2, si on considère que les variables qui constituent les cibles potentielles peuvent être modifiées sur l'ensemble de leur plage de valeurs, il est possible de calculer pour un programme donné le taux théorique de détection aux différents points d'injection. Prenons pour exemple un appel de fonction qui ne dépend que d'une seule variable de type *char*. Cette variable peut donc prendre 256 valeurs différentes possibles (un *char* contient une valeur d'une taille de 8 *bits*). Parmi ces 256 valeurs, une seule valeur est valide à un instant t. Cela nous fait donc $256 - 1$ valeur possibles pour l'injection soit 255 valeurs.

Supposons que l'analyse de valeur a déterminé qu'à ce point précis du programme, cette variable ne peut prendre que 24 valeurs différentes possibles. Nous avons donc $24 - 1$ valeurs d'injection possibles qui ne peuvent être détectées, soit 23 valeurs. Si la valeur à injecter est choisie aléatoirement parmi les 255 valeurs possibles pour l'injection et ce de manière équiprobable, alors dans ce cas le taux de faux négatifs de notre approche pour la détection est de $23 \div 255$ soit 9%. Si par contre l'analyse de valeur n'a pas pu déterminer statiquement le domaine de variation de la variable, alors dans ce cas le taux de faux négatifs est de 100%.

Toutefois, ce type de calcul ne tient compte de l'impact de la corruption qu'au point du programme où l'injection a lieu. Or il est possible que l'erreur se propage à d'autres variables. Une injection peut donc avoir un impact sur d'autres variables plus loin dans l'exécution du programme. Ceci pourrait permettre à notre système de détection d'intrusion de lever une alerte bien après le point d'injection grâce à des invariants portant sur d'autres variables que celle qui a été la cible de l'injection. Ce type d'alertes est utile lorsque l'attaque cible une variable qui ne peut voir son domaine de variation calculé statiquement pour le point du programme où l'injection est réalisée.

En effet, dans ce cas, il n'est bien sûr pas possible de détecter cette injection immédiatement. Cependant, si la corruption de cette variable particulière provoque par propagation de l'erreur la corruption d'une autre variable, il existe donc une possibilité que l'injection soit détectée par notre mécanisme de détection plus tard dans l'exécution du programme. Ceci augmente donc les chances qu'une simulation d'attaque soit détectée durant l'exécution du programme. Si nous réalisons une campagne d'injection pour un programme donnée, c'est parce que nous voulons obtenir une estimation du taux de faux négatifs qui, certes, soit une sur-approximation du taux réel, mais qui néanmoins prenne en compte la problématique de l'impact. Nous allons donc maintenant voir dans cette section ce qui peut empêcher ou limiter l'impact d'une injection.

Tout d'abord, il est possible que l'erreur engendrée par une injection soit annulée ce qui a pour effet de rendre à nouveau correct l'état interne du processus en cours d'exécution. Cela peut se produire dans deux situations. D'abord, il est possible que plusieurs erreurs interagissent entres elles de manière à compenser leurs effets

respectifs. Par exemple, une première erreur va inverser un bit dans une variable qui ne sera plus modifiée par le programme jusqu'à ce que se produise un peu plus tard une seconde erreur qui va avoir le même effet sur la variable ce qui aura pour conséquence de restaurer le bit précédemment corrompu.

Comme nous le verrons dans la section 3.1.2.1, nous ne procédons qu'à une seule injection par exécution du programme. Ce cas de figure ne peut donc pas se présenter avec notre modèle. L'autre situation qui a pour effet de corriger un état interne erroné peut se produire lorsque des instructions exécutées réinitialisent la valeur des cibles de l'injection. Par exemple, dans le cas où l'injection cible une seule variable, l'erreur est annulée lorsque la valeur de la variable corrompue est réinitialisée avant d'avoir pu propager cette erreur. Dans ce situation, l'éventuel impact de l'erreur sur le programme est limité au code exécuté entre le point d'injection et le code de réinitialisation de la variable.

Ensuite, dans le cas où l'erreur persiste durant tout ou partie de l'exécution du programme, il est possible que le code exécuté masque l'état erroné du processus. Cela peut se produire dans plusieurs situations. Tout d'abord, la plus simple : bien que la variable ciblée par l'injection ait influencée le code exécuté jusqu'au point d'injection, la valeur de celle-ci n'est déjà plus utilisée par le programme (soit la variable n'est plus du tout utilisée, soit sa valeur est réinitialisée avant toute nouvelle utilisation). Si la valeur injectée n'est plus utilisée, elle ne peut donc pas propager l'erreur dans le programme.

Autre situation, la variable ainsi corrompue provoque une déviation du flot de contrôle mais celle-ci n'engendre pas de déviation du flux d'information au sein de ce flot d'exécution non prévu. C'est-à-dire que le chemin d'exécution se comporte exactement comme prévu s'il n'avait pas été choisi illégalement. Par exemple, la variable ciblée par l'injection va modifier le résultat d'une évaluation conditionnelle mais le nouveau chemin d'exécution ainsi choisi n'utilise pas la valeur de cette variable. Le code ainsi exécuté, bien qu'illégal, ne permettra pas de détecter l'état erroné du programme.

De plus, il existe aussi trois situations où, bien que la valeur injectée soit utilisée par le programme, celle-ci ne provoque aucune déviation ni du flot de contrôle ni du flot de donnée. D'abord, il est possible que la nouvelle valeur ne change pas le résultat de l'évaluation d'une expression arithmétique. Considérons par exemple l'expression suivante : $x \times y$. Si la variable x est la cible de l'injection et qu'au moment d'évaluer l'expression la variable y vaut 0, alors quelque soit la valeur injectée celle-ci ne modifiera pas le résultat de l'évaluation.

De même, il est possible que la nouvelle valeur ne change pas le résultat de l'évaluation d'une expression logique. Considérons par exemple l'expression suivante : $x \vee y$. Si l'injection modifie le résultat de l'expression logique x mais que l'expression logique y est vraie, alors quelle que soit la valeur injectée celle-ci ne modifiera pas le résultat de l'évaluation de l'expression dans sa globalité. Enfin, il est possible que la nouvelle valeur ne change pas le résultat de l'évaluation d'une expression relation-

nelle. Considérons par exemple l'expression suivante : $x > y$. Si la variable x est la cible de l'injection et que la nouvelle valeur appartient à la même classe d'équivalence que la valeur originale, alors le résultat de l'évaluation restera inchangé.

3.1.1.3 Détection et caractérisation des erreurs

Nous allons maintenant aborder la problématique de savoir dans quelle situation le système de détection d'intrusion que nous proposons au chapitre 2 est bien en mesure de détecter une injection. Lorsque que nous injectons une valeur incorrecte au sein d'un ensemble de variables juste avant un appel de fonction, deux cas se présentent. Pour commencer, supposons que le domaine de variation de cet ensemble de variables a pu être calculé au point d'injection et que l'état interne du programme une fois corrompu place effectivement l'ensemble de variables hors de ce domaine de variation. Dans ce cas, l'injection est effectivement détectée immédiatement. Par contre, à l'inverse, si un tel domaine de variation n'a pu être calculé ou bien si l'état interne du programme, bien que corrompu, place l'ensemble de variables dans un état invalide mais néanmoins vraisemblable, dans ce cas l'injection ne sera pas immédiatement détectée.

Toutefois, même lorsque que ce second cas se présente, ceci n'implique pas nécessairement que l'injection ne sera jamais détectée par notre mécanisme de détection d'intrusion. En effet, l'injection ne sera pas détectée immédiatement, mais comme nous avons pu le voir dans la section précédente, il est possible que l'erreur introduite au point d'injection se propage à d'autres variables. Dans le cas où celles-ci appartiendraient à des ensembles de variables dont nous avons pu calculer le domaine de variation plus loin dans l'exécution du programme, il est alors possible que l'injection soit détectée ultérieurement. Cela peut par exemple se produire si l'injection qui n'est pas immédiatement détectée place la fonction en cours d'exécution dans un état de défaillance. Le résultat erroné retourné par cet appel de fonction a la possibilité de propager l'erreur à d'autres variables dont le domaine de variation est connue (voir figure 3.1).

À noter que la détection n'est pas liée à la problématique de l'impact. En effet, nous avons aussi vu dans la section précédente qu'il existe des situations où bien que l'état interne du programme soit erroné, le code exécuté masque les erreurs ainsi introduites. Cela n'empêche en rien la détection. Par exemple, bien qu'une erreur soit masquée par le fait que la variable corrompue ne va plus être utilisée par le programme, si on sait qu'à ce point de l'exécution elle ne peut valoir qu'une valeur bien précise, alors l'injection sera tout de même détectée.

Par contre, la problématique de l'impact va avoir des conséquences sur notre capacité à détecter une défaillance. Rappelons que l'objectif d'une campagne de tests à l'aide du mécanisme d'injection que nous proposons est d'évaluer notre système de détection d'intrusion. On peut cependant se poser la question de la pertinence des résultats obtenus pour la simulation d'attaques contre les données de calcul

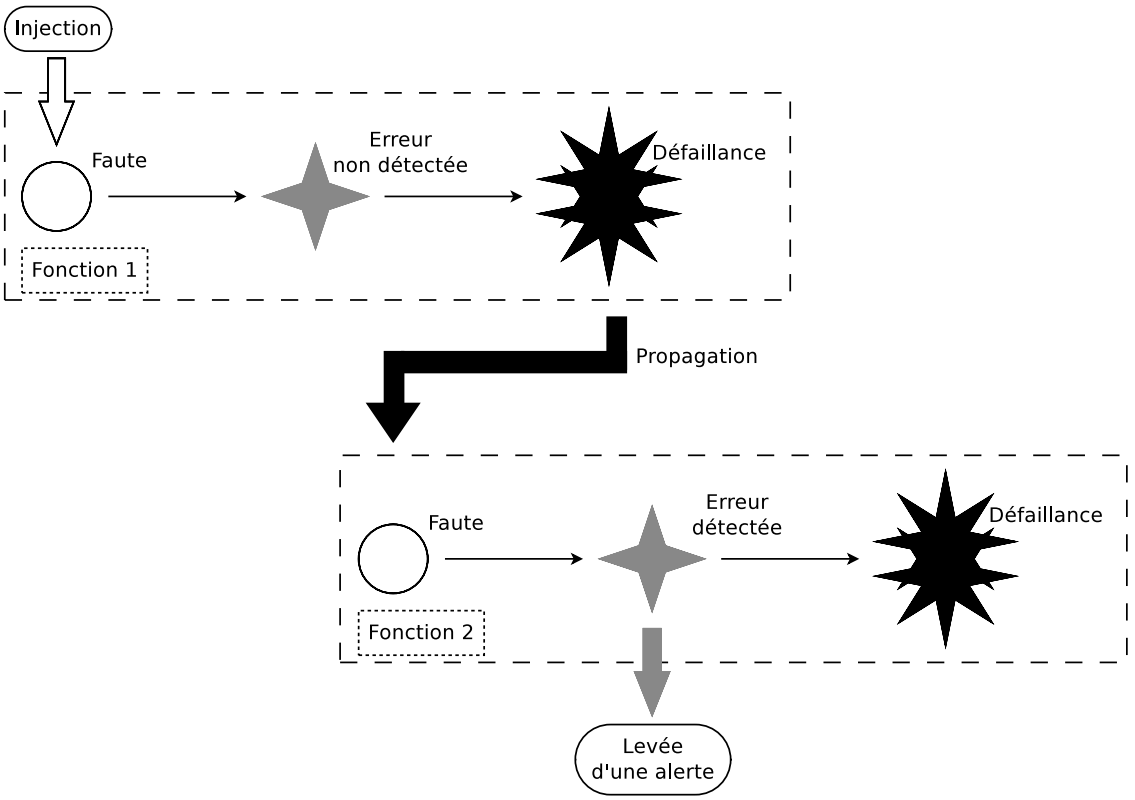


FIGURE 3.1 – Détection de l'injection par propagation

à l'aide de notre modèle de faute par rapport d'attaques réelles. C'est pourquoi nous proposons de caractériser le comportement de l'application en fonction de ses séquences d'appels système mais aussi en fonction des données de sortie produites suite à l'exécution de scénarios bien précis. Ainsi, lorsque nous observons une déviation comportementale de l'application suite à l'injection et que celle-ci n'est pas détectée, alors nous pourrons conclure qu'il s'agit forcément d'une attaque qui n'a pas été détectée.

Toutefois, si aucune déviation comportementale n'est observée, nous ne pouvons conclure qu'il ne s'agit pas d'une attaque possible. En effet, même sans déviation observable au niveau des appels système ou des données produites par le programme, une injection peut reproduire le résultat d'une attaque contre les données de calcul. Prenons à nouveau comme exemple le pseudo-code de la figure 2.3 que nous avons utilisé au chapitre 2 pour présenter notre modèle de détection. La seconde attaque, qui consiste à forcer la valeur de la variable *ok* lorsque le message reçu n'est pas de type $CMSG\_AUTH\_PWD$ peut être simulée juste avant l'appel à la fonction $do\_authenticated$ en modifiant la variable *type*. Cependant, il s'agit là d'une erreur masquée puisque celle-ci n'est plus utilisée par le programme. Aucune déviation comportementale ne sera donc observée au niveau des appels système comme des

données sorties par le programme alors même que l'injection simule bien le résultat de cette attaque particulière.

Au final, lorsqu'une déviation comportementale est observé, l'ensemble des injections non détectées constitue un nombre de cas sûr durant lesquels notre système de détection d'intrusion est en situation d'échec. Bien que cette mesure apporte de l'information quant à l'évaluation de notre approche, il faudra cependant considérer l'ensemble des injections non détectées par notre approche pour obtenir une sur-approximation du taux réels de faux négatifs.

3.1.1.4 Construction du modèle d'injection

Pour pouvoir construire de manière automatique notre modèle de faute pour un programme donné, nous devons être capable de déterminer quels sont les éléments mémoire qui constituent l'ensemble des cibles potentielles à injecter et pour chacun d'entre elles, quand est-ce qu'il est pertinent de procéder à l'injection. Nous allons donc maintenant aborder ces deux problématiques.

Tout d'abord, abordons celui des cibles potentielles à injecter. Il s'agit bien sûr ici de déterminer au sein de l'état interne du programme les éléments qui constituent le sous-ensemble des données qui peuvent être considérées comme une cible intéressante par un utilisateur malveillant. Nous avons vu dans la section 3.1.1.1 où nous présentons les caractéristiques à simuler qu'il s'agit dans le cas d'attaques contre les données de calcul des éléments mémoire qui contiennent la valeur des variables dont dépendent les appels système. Or, nous avons aussi vu au chapitre précédent dans la section 2.3.2 que si l'on dispose du code source du programme, alors cet ensemble peut être déterminé par analyse statique et plus précisément en utilisant des coupes de programmes. Nous proposons donc d'avoir à nouveau recours à l'analyse statique pour déterminer l'ensemble des cibles potentielles à injecter.

Maintenant que nous connaissons cet ensemble de données, nous devons déterminer pour chacune d'entre elles à quel moment réaliser l'injection. Pour que l'injection soit pertinente avec l'objectif de simuler une attaque contre les données de calcul, celle-ci doit avoir lieu lorsque la donnée que l'on cible contient une valeur dont dépend un appel système qui n'a pas encore été exécuté même si la valeur en question a déjà été utilisée. Puisque l'on dispose du code source du programme et que les données que l'on souhaite injecter sont identifiables par le nom des variables dont elles contiennent la valeur, le plus simple est d'instrumenter le code source pour y placer avant chaque appel système un mécanisme d'injection contrôlable depuis l'extérieur du programme.

Là encore, nous faisons face aux mêmes problèmes d'accessibilité qu'à la section 2.3.2, à savoir que les appels système ne sont pas nécessairement tous présents dans le code source et l'ensemble des variables dont dépend un appel système donné ne sont pas nécessairement toutes accessibles dans le code source au moment de l'exécution de l'appel. À nouveau, pour résoudre ces problèmes, nous choisissons de

déplacer le problème du calcul des coupes de programme des appels système aux appels de fonction et de distribuer les mécanismes d'injection dans le code source. Ainsi, avant chaque appel de fonction présent dans la pile d'appel menant à un appel système et accessible depuis le code source, un mécanisme d'injection est ajouté. Celui-ci est capable de modifier la valeur de n'importe quelle variable accessible à ce point du programme et dont dépend l'appel de fonction concerné.

3.1.2 Évaluation du taux de détection

Maintenant que nous savons construire un modèle de faute permettant de simuler les caractéristiques des attaques contres les données de calcul, nous devons aborder la problématique suivante : comment utiliser ce modèle de faute dans le cadre de l'évaluation d'un système de détection d'intrusion ? En effet, pour l'instant le mécanisme d'injection qui utilise le modèle de faute que nous proposons dans ce chapitre reste relativement générique. Certes il est conçu pour ne viser que les cibles potentielles pour un utilisateur malveillant cherchant à réaliser une attaque contre les données de calcul, mais il n'impose ni le nombre d'injections qui doivent être réalisées à chaque exécution, ni le nombre de cibles à corrompre à chaque injection, ni la valeur à injecter pour chaque cible.

Pour répondre à cette problématique, des choix doivent donc être faits. Ce sont ces choix que nous allons présenter dans la suite de cette section. Pour comprendre la démarche que nous avons entreprise pour nous guider dans ces choix, il faut se rappeler que ces décisions ont été prises sous la contrainte d'une exigence bien particulière : nous cherchons à placer le système de détection d'intrusion en cours de test dans la situation qui lui est la moins favorable. Ceci nous assurera une surapproximation de l'estimation du taux de faux négatifs. À noter que pour répondre à ce critère, les choix ont été faits en fonction spécifiquement de notre approche pour la détection et ces derniers pourraient être totalement différents pour l'évaluation d'un tout autre mécanisme de détection.

3.1.2.1 Choix du nombre d'injections

Nous allons maintenant déterminer combien d'injections doivent être réalisées durant l'exécution du programme. Pour cela, nous devons considérer deux cas. Soit l'attaquant a la possibilité de modifier toutes les variables nécessaires à la réalisation de l'intrusion en une seule fois, soit il lui est nécessaire d'exploiter la vulnérabilité à plusieurs reprises. C'est dans le second cas que la probabilité de détection par notre mécanisme est la plus élevée.

En effet, plus l'attaque nécessite d'exploiter la ou les vulnérabilités présentes dans le programme plus grande est la quantité de code exécuté dans un état illégal et donc plus grande est la probabilité de violer au moins un invariant. Afin de nous placer dans la situation la moins favorable pour le mécanisme de détection que nous

proposons, nous choisissons donc d'effectuer la simulation d'une attaque contre les données de calcul en une seule injection.

3.1.2.2 Choix du nombre de cibles

Nous devons aussi déterminer combien de variables à la fois doivent être corrompues par une injection. Nous faisons ici, dans le cas de notre approche pour la détection, l'hypothèse suivante : plus une injection modifie de variables à la fois, plus elle a de chance de provoquer la levée d'alertes par notre système de détection d'intrusion. Nous allons donc vérifier cette hypothèse sur un exemple pour notre système de détection d'intrusion.

Pour cela, nous allons d'abord poser un certain nombre de notations et de définitions. Soient n variables locales à une fonction, notées X_1, X_2, ..., X_n, dont les domaines de définition sont respectivement les ensembles D_1, D_2, ..., D_n. On suppose que, pour un point donné de l'exécution du programme, nous pouvons déterminer par analyse statique un ensemble $V \subseteq D_1 \times D_2 \times \ldots D_n$ qui correspond à l'ensemble des vecteurs possibles du n-uplet constitué par les n valuations des variables X_1, ..., X_n au point considéré. Cet ensemble est appelé domaine de variation et il s'agit d'un sur-ensemble de S, l'ensemble des n-uplets valides du point de vue de la spécification du programme analysé.

Cela signifie que pour toute exécution de ce programme, lors du passage au point d'exécution choisi, le vecteur $x = (x_1, \ldots, x_n)$ constitué par les valuations des variables X_1, X_2, ..., X_n est un élément de $S \subseteq V$. Le mécanisme de détection d'intrusion que nous proposons insère avant le point d'exécution considéré des assertions exécutables visant à vérifier que le n-uplet constitué par les n valuations des variables X_1, ..., X_n forme bien un élément de l'ensemble V. Dans le cas contraire une alerte est levée.

Prenons comme exemple le cas simple où $n = 2$ et où, au point considéré, la variable X_1 a pour domaine de variation I_1 un intervalle continu sur D_1 et la variable X_2 a pour domaine de variation I_2 un intervalle continu sur D_2. Nous pouvons en déduire que $V = I_1 \times I_2$ représente l'ensemble des valeurs possibles pour le vecteur $x = (x_1, x_2)$. Cet ensemble est représenté par le rectangle central de la figure 3.2(a). Lorsqu'une injection modifie seulement la valeur de x_1, une alerte est levée si la nouvelle valeur n'est pas dans l'ensemble I_1 (tandis que la valeur de x_2 reste dans l'ensemble I_2). De même, lorsqu'une injection modifie seulement la valeur de x_2, une alerte est levée si la nouvelle valeur n'est pas dans l'ensemble I_2 (tandis que la valeur de x_1 reste dans l'ensemble I_1).

Au final, l'ensemble des injections qui provoquent la levée d'une alerte correspond à l'ensemble $(D_1 \times I_2) \cup (D_2 \times I_1) \setminus (I_1 \times I_2)$. Cet ensemble est représenté par l'espace de couleur gris foncé de la figure 3.2(b). Lorsqu'une injection modifie à la fois les valeurs de x_1 et de x_2, une alerte est levée si le nouveau couple de valeurs n'est pas dans l'ensemble $I_1 \times I_2$. Cette fois, l'ensemble des injections qui provoquent la levée

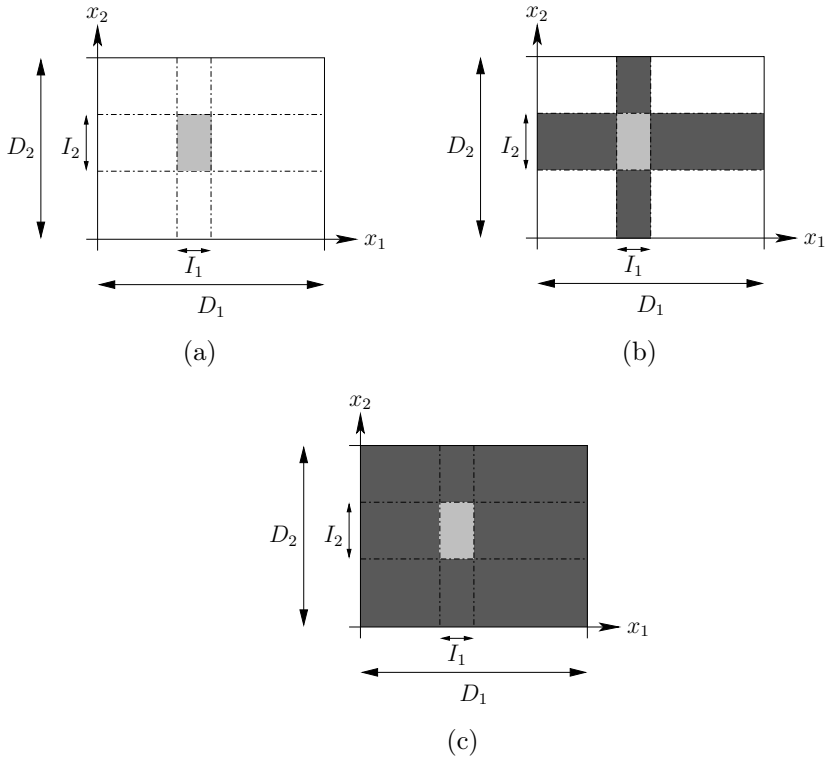


FIGURE 3.2 – Domaine de détection d'une injection à deux variables

d'une alerte correspond à l'ensemble $(D_1 \times D_2) \setminus (I_1 \times I_2)$ Cet ensemble est représenté par l'espace de couleur gris foncé de la figure 3.2(c).

Si maintenant nous comparons ces deux situations, nous pouvons voir que le dernier ensemble, celui qui correspond à l'injection de deux variables simultanément, inclut le précédent ensemble, celui qui correspond à l'injection d'une seule variable à la fois. Ainsi, sur cet exemple, en augmentant le nombre de variables modifiées par une injection, on augmente effectivement la surface de détection de notre approche, et dont on réduit le taux de faux négatifs de notre système de détection d'intrusion. Au final, cet exemple illustre bien le fait qu'afin de placer notre approche pour la détection dans la situation la moins favorable, il ne faut donc modifier qu'une seule variable par injection. Une preuve de cette affirmation pour n et V quelconques est disponible en annexe.

3.1.2.3 Choix du type de corruption

Maintenant que nous savons que nous n'allons réaliser qu'une seule injection par exécution du programme à tester et que cette injection ne va cibler qu'une seule donnée, il nous faut choisir la valeur à fournir au mécanisme d'injection pour la corruption de la variable. Lors des situations réelles d'attaque que nous avons

présentées dans la section 2.3.1.1, les valeurs utilisées pour une attaque donnée sont liées au fonctionnement interne du programme ciblé. Il nous est donc impossible de connaître ces valeurs de manière automatique.

De plus, nous avons vu dans la section 3.1.1.4 que pour respecter la contrainte de ne simuler que des attaques réalisables grâce à une seule et unique exploitation de la vulnérabilité présente dans le programme, nous ne pouvons pas tenir compte de la valeur initialement contenue par la donnée ciblée. C'est pourquoi nous faisons le choix de ne pas émettre d'hypothèse sur ces dernières et de les tirer aléatoirement parmi la plage de valeur de l'élément ciblé lors de l'injection. Au final, lors d'une injection, nous allons donc choisir aléatoirement la nouvelle valeur parmi toute la plage de valeur possible de l'élément ciblé et ce sans tenir compte de la valeur originalement contenue.

3.1.2.4 Avantages et limites du modèle

Nous avons choisi de simuler la situation la moins favorable pour notre système de détection d'intrusion : nous allons simuler la présence en tout point du programme d'une vulnérabilité permettant un accès direct à l'ensemble de l'espace mémoire du processus en cours d'exécution et les attaques qui exploitent cette vulnérabilité n'ont besoin pour être réalisées que d'une seule exploitation et de ne corrompre qu'une seule donnée. Le modèle de faute que nous proposons dans ces travaux de thèse permet effectivement de simuler le résultat d'attaques contre les données de calcul ayant ce niveau de sévérité.

Cependant, nous avons pu voir au chapitre précédent dans la section 2.3.1.1, où nous illustrions le déroulement d'une attaque contre les données de calcul, que la corruption résultant de l'exploitation d'une vulnérabilité peut aussi bien nécessiter une valeur quelconque (cas de l'attaque contre la variable de boucle *ok*) qu'une valeur bien précise (cas de l'attaque contre la variable *pwd*). Nous n'avons pas moyen de déterminer automatiquement pour chacune des cibles si toute la plage de valeur est représentative d'une attaque contre les données de calcul ou bien si seule une liste de valeurs bien précises est pertinente. Notre modèle de faute ne permet donc pas à coup sûr de simuler une attaque nécessitant de corrompre sa cible avec une valeur bien précise.

Toutefois, plus nous effectuons de simulations d'attaque à l'aide de notre modèle de faute, plus nous avons de chance de simuler des états internes erronés représentatifs des situations réelles d'attaques contre les données de calcul. Afin de compléter l'évaluation de notre mécanisme de détection d'intrusion, nous proposons également d'estimer le taux de faux négatifs pour les simulations d'attaques qui ont sans le moindre doute provoqué une déviation comportementale.

Pour déterminer cela, nous proposons d'analyser les traces d'exécution du programme en cours de test. Cette analyse des résultats peut effectivement nous permettre d'identifier des simulations d'attaque qui sont représentatives des situations

réelles d'attaques contre les données de calcul. Mais en aucun cas cette analyse ne peut nous permettre de toutes les identifier. Cela est dû au fait qu'une simulation d'attaque, bien que représentative d'une attaque réelle, ne provoque pas nécessairement une déviation comportementale observable (voir la problématique du masquage d'erreurs à la section 3.1.1.3). Nous aborderons ceci plus en détails dans la section 4.2.2.1 où nous présentons l'outil que nous avons utilisé pour mettre en évidence les déviations comportementales produites par les injections.

3.2 Implémentation du mécanisme d'injection

Pour tester notre approche pour l'évaluation et afin de l'appliquer à notre modèle de détection présenté au chapitre 2, nous avons implémenté le modèle d'attaque que nous présentons dans ce chapitre. L'implémentation de notre approche pour la détection cible les programmes écrits en langage *C*. C'est donc également le cas de l'implémentation de notre approche pour l'évaluation. Là encore, nous voulons que le processus de création du modèle de faute et le processus de génération des mécanismes d'injection soient entièrement automatiques.

Comme nous l'avons vu à la section 3.1.1.4, les techniques d'analyse statique utilisées au chapitre 2 pour construire notre modèle de détection peuvent aussi être utilisées pour construire notre modèle d'injection. L'outil que nous avons présenté dans la section 2.4.2.1 a été modifié afin de procéder également à l'instrumentation de programmes écrits en langage *C* pour réaliser des injections de faute selon le modèle décrit dans ce chapitre.

Nous allons maintenant présenter comment nous avons implémenté la construction de notre modèle d'injection de fautes au sein de notre outil et comment les mécanismes d'injection générés à partir de ce modèle peuvent être contrôlés de l'extérieur du programme durant son exécution. Pour cela, nous abordons d'abord la question de l'instrumentation du programme à l'aide de mécanismes d'injection déduits de notre modèle de faute. Puis, nous présentons en détail le déroulement de la procédure d'injection en expliquant le fonctionnement des mécanismes d'injection que nous avons choisis pour évaluer notre système de détection d'intrusion et comment ces derniers peuvent être contrôlés par un moniteur externe.

3.2.1 Instrumentation du programme

Avant d'avoir recours à un mécanisme d'injection, il faut tout d'abord d'avoir déterminé le modèle de faute qui va être utilisé durant la campagne de tests. Il peut s'agir d'un modèle simple et qui ne nécessite aucun calcul. C'est par exemple le cas d'un modèle cherchant à corrompre de manière complètement aléatoire l'ensemble de l'espace mémoire d'un processus. Mais cette étape peut aussi être complexe et nécessiter d'avoir recours à une phase d'analyse. C'est le cas du modèle de faute que nous proposons dans ce chapitre.

Notre modèle de faute, une fois calculé, va être utilisé avec un mécanisme d'injection qui repose sur l'instrumentation des programmes. Dans le cas du mécanisme d'injection que nous proposons dans ces travaux, l'instrumentation dépend en partie de notre modèle de faute. La phase d'instrumentation ne peut donc être réalisée qu'une fois le modèle de faute connu.

Plus précisément, ce dont nous avons besoin de calculer pour obtenir notre modèle de faute pour un programme donné et qui est nécessaire à l'instrumentation de ce programme, c'est l'ensemble des cibles potentielles pour un utilisateur malveillant voulant réaliser une attaque contre les données de calcul. Comme nous l'avons vu à la section 3.1.1.1, il s'agit pour chaque appel de fonction menant à un appel système de l'ensemble des variables localement accessibles et dont il dépend. Puis, une fois ces ensembles connus, nous pouvons passer à la phase d'instrumentation. Il s'agit ici de déduire de ces ensembles de variables les mécanismes d'injection et de les ajouter au code source. Nous allons maintenant détailler ces deux étapes.

3.2.1.1 Calcul de l'ensemble des cibles

Comme précédemment, nous choisissons de reposer sur la coupe de programme et plus précisément sur le graphe de dépendance du programme [KKP+81, FOW87] afin de déterminer pour chaque appel de fonction l'ensemble des variables dont il dépend et qui sont localement accessibles au moment de son exécution. Nous réutilisons donc les résultats du greffon *Program Dependence Graph* que nous utilisions jusqu'à maintenant pour générer les mécanismes de détection. Les ensembles de variables ainsi obtenus vont ensuite être utilisés pour générer le code d'injection propre à chaque appel de fonction. La génération des mécanismes d'injection nécessite aussi qu'on lui fournisse une fonction de traitement des demandes d'injections. Dans la suite de ce chapitre, nous la désignerons simplement comme la fonction d'injection. Nous détaillons dans la section suivante celle que nous avons utilisée dans le cadre de l'évaluation de notre système de détection d'intrusion.

Toutefois, notons que lors de la construction du modèle de détection, une fois l'ensemble des cibles potentielles déterminé, seul le sous-ensemble de variables dont le domaine de variation a pu être calculé est ensuite contrôlé par le mécanisme de détection. Dans le cadre de l'évaluation d'un système de détection d'intrusion, notre mécanisme d'injection ne requiert pas cette information. Cela veut dire que si nous appliquons notre méthode d'évaluation au mécanisme de détection que nous proposons, ce sont bien l'ensemble des variables dont dépendent les appels système qui vont être la cible d'une corruption durant la campagne d'injection et pas seulement celles que notre mécanisme de détection est capable de vérifier.

3.2.1.2 Ajout des mécanismes d'injection

Comme précédemment au chapitre 2, les fichiers sources doivent être pré-traités avant analyse afin de tenir compte des variables supplémentaires ajoutées par *Frama-*

C [FRAMA] lors de la construction du modèle de faute mais aussi pour ajouter les entêtes nécessaires aux mécanismes d'injection qui vont être générés suite à cette analyse. De même, comme précédemment, une fois les mécanismes d'injection obtenus, nous générons un ensemble de correctifs pour le code source du programme. Ces correctifs contiennent pour chaque fichier du code source qui doit être modifié les mécanismes d'injection qui lui sont propres.

Notons que puisque notre outil est maintenant capable de générer à la fois des mécanismes de détection et des mécanismes d'injection, il nous faut considérer le cas où, pour un appel de fonction donné, les deux types de mécanismes doivent être insérés juste avant l'instruction. Dans ce cas, nous avons choisi de placer le mécanisme d'injection avant le mécanisme de détection. Ceci nous permet de tester notre approche pour la détection durant l'exécution du programme dès le point d'injection. La figure 3.3 présente sur un exemple les mécanismes d'injections ajoutés par la phase d'instrumentation (il s'agit des appels à la fonction *inject*).

À l'instar des mécanismes de détection, les paramètres des mécanismes d'injection correspondent aux paramètres de la fonction chargée de répondre aux demandes d'injection. Notons que les mécanismes d'injection ont également besoin d'une information supplémentaire par rapport aux mécanismes de détection : la taille des variables. Cette information permet de s'assurer que la valeur à injecter et qui est fournie par un contrôleur externe ne dépasse pas la taille de la variable ciblée. En effet, les mécanismes d'injection ont accès aux variables ciblées via des pointeurs non-typés. La connaissance de la taille des variables nous permet de garantir que l'injection ne déborde pas sur d'autres variables et nous permet ainsi de respecter les contraintes exprimées dans les sections 3.1.1.1 et 3.1.2.

L'ensemble des paramètres de chaque mécanisme d'injection varie selon l'emplacement et les dépendances de l'appel de fonction auquel il se rapporte. Le premier paramètre de l'appel est un identifiant unique. Il s'agit du numéro de l'instruction dans l'arbre syntaxique abstrait utilisé par *Frama-C* [FRAMA]. Le second paramètre de l'appel correspond au nombre de variables qui peuvent être la cible d'une injection à ce point précis du programme. Enfin, les derniers paramètres correspondent à l'adresse de chacune de ces variables en question ainsi qu'à la taille de la donnée contenue par celles-ci.

3.2.2 Déroulement de la procédure

Le déroulement d'une procédure d'injection dépend du comportement de la fonction d'injection fournie durant la phase d'instrumentation mais aussi des paramètres qui lui sont communiqués depuis l'extérieur du programme. La fonction d'injection que nous fournissons par défaut avec notre outil est celle que nous avons utilisée pour tester notre système de détection d'intrusion.

La manière dont cette fonction gère les demandes d'injection va avoir des conséquences sur les données que nous allons pouvoir récolter pour analyser *a posteriori*

```
00. void init(int c, int v, int o)
01. {
02.   char p;
03.
04.   if (v == 0) {
05.     p = 1;
06.   }
07.   else {
08.     p = -1;
09.   }
10.
11.   if (c == 0) {
12.
13.
14.
15.     func(p);
16.   }
17.   else if (c == 1) {
18.     p *= o;
19.
20.
21.
22.
23.     func(p);
24.   }
25. }
```

```
00. void init(int c, int v, int o)
01. {
02.   char p;
03.
04.   if (v == 0) {
05.     p = 1;
06.   }
07.   else {
08.     p = -1;
09.   }
10.
11.   if (c == 0) {
12.     inject(0x00000015, 2,
13.            &c, sizeof(int),
14.            &p, sizeof(char));
15.     func(p);
16.   }
17.   else if (c == 1) {
18.     p *= o;
19.     inject(0x00000023, 3,
20.             &c, sizeof(int),
21.             &o, sizeof(int);
22.             &p, sizeof(char));
23.     func(p);
24.   }
25. }
```

FIGURE 3.3 – Exemple d'ajout de code d'injection

le comportement de notre système de détection d'intrusion pendant la campagne de tests. C'est pourquoi nous détaillons son fonctionnement interne dans la suite de cette section. La manière dont les paramètres d'injection sont communiqués à la fonction d'injection a quant à elle des conséquences sur la manière dont doit être implémenté le contrôleur externe. Pour finir, nous allons donc aborder dans cette section la manière dont cette fonction doit être paramétrée de l'extérieur du programme.

3.2.2.1 La fonction d'injection

L'outil que nous avons développé laisse le choix quant à l'implémentation du mécanisme d'injection. L'utilisateur peut ainsi fournir sa propre procédure d'injection tant que celle-ci respecte le prototype présenté dans la section précédente. Toutefois, notre outil contient une implémentation par défaut conçue pour être contrôlable via un processus externe. C'est cette dernière que nous avons utilisée pour tester notre système de détection d'intrusion et nous allons maintenant détailler son fonction-

nement interne.

Nous avons énoncé dans la section 3.1.2.1 que nous ne voulons effectuer qu'une seule corruption de donnée lors de chaque exécution du programme. Le fait que les mécanismes d'injection soient embarqués dans le programme peut alors poser problème : que se passe-t-il lorsqu'un mécanisme d'injection est placé dans une portion de code exécuté en boucle ou tout simplement dans une fonction appelée plusieurs fois durant le déroulement du scénario d'exécution. La fonction d'injection fournie par défaut avec notre outil utilise une variable d'environnement pour savoir si elle a déjà effectué une injection ou non. Ainsi, celle-ci nous assure que, quelque soit l'emplacement des mécanismes d'injection, la corruption de la variable choisie comme cible ne peut avoir lieu qu'une et une seule fois.

Afin de pouvoir analyser *a posteriori* les résultats obtenus suite à la campagne de tests, la fonction d'injection doit communiquer un certain nombre d'informations. La plus importante d'entre elles est bien sûr de savoir si le mécanisme d'injection dont nous avons demandé l'activation s'est bien activé comme prévu. Cette information est envoyée sur la sortie erreur du programme et lorsque tel est le cas cette dernière rappelle également, en plus de l'identifiant du mécanisme d'injection ainsi activé, la variable choisie pour cible et la valeur injectée. L'ensemble de ces informations pourra permettre au besoin de reproduire une injection particulière, par exemple lorsque celle-ci aura effectivement provoqué une déviation comportementale.

Notons que ce signalement n'a pas lieu si la valeur à injecter qui est fournie en paramètre est égale à la valeur originalement contenue par la variable ciblée. La fonction d'injection effectue cette vérification afin de nous assurer que les injections mettent nécessairement le programme dans un état interne non prévu. En effet, nous avons vu à la section 3.1.2 que nous injectons une variable sur toute sa plage de valeur sans connaissance de sa valeur original. La situation où nous injectons la même valeur que celle déjà contenue par la variable peut donc se produire. Lorsque que cette absence de signalisation se produit, le moniteur pourra alors ignorer ce test et recommencer l'injection avec une nouvelle valeur aléatoire. Nous avons fait le choix de signaler l'absence d'injection plutôt que de donner à la fonction d'injection la capacité de modifier la valeur à injecter afin de ne pas contourner la capacité de contrôle du moniteur.

Toutefois, chaque mécanisme d'injection, que celui-ci soit activé ou non, se signale lorsque le chemin d'exécution passe par lui. Cette information est également envoyée sur la sortie erreur et rappelle l'identifiant du mécanisme d'injection qui se signale. Ce type d'information permet plusieurs choses. Tout d'abord, cela permet de connaître l'ensemble des mécanismes d'injection qui sont atteignables pour un scénario donné. Pour obtenir cela, il suffit d'exécuter ce scénario sans activer de mécanisme d'injection. Il est nécessaire de connaître cela pour l'ensemble des scénarios et ce avant la phase de tests afin de pouvoir choisir un scénario qui permet d'atteindre un mécanisme d'injection particulier.

Enfin, ce type d'information peut nous permettre d'estimer la distance d'exécu-

tion entre le point du code où l'injection a lieu et les différents points du code où une erreur est détectée par notre mécanisme. En effet, en tenant compte des mécanismes d'injections qui se signalent après que l'injection soit effective, nous pouvons connaître le nombre d'appels de fonction exécutés entre la corruption de la variable ciblée et les différents moments où un invariant n'est pas vérifié. Nous avons vu dans la section 3.1.1.2, où nous abordons la question de la propagation des erreurs, que le taux de détection au point d'injection est inférieur ou égal au taux de détection sur l'ensemble du programme. Avec cette mesure, nous pouvons juger de la pertinence d'une campagne de tests par rapport au calcul du taux de détection théorique aux points d'injection.

À noter que peut se présenter le cas où la phase d'instrumentation n'a pas réussi à déterminer d'ensemble de variables pour un appel de fonction donné. Dans ce cas, ce dernier ne possède donc pas de mécanisme d'injection. Afin que notre connaissance des appels de fonction exécutés après le point d'injection soit correcte, nous avons modifié notre outil pour qu'une simple instruction de signalement soit placée avant un appel de fonction lorsque cette situation se présente.

3.2.2.2 Le paramétrage de l'injection

Afin de pouvoir s'adapter à différentes procédures d'évaluation, les mécanismes d'injection générés par défaut sont contrôlables de l'extérieur du processus. Le contrôle s'effectue grâce à des variables d'environnement et celles-ci permettent de choisir quel mécanisme d'injection doit être activé, sur quelle variable la corruption doit être effectuée et enfin quelle est la nouvelle valeur que doit contenir la variable après l'injection.

Dans le cas de la fonction d'injection que nous utilisons, il s'agit respectivement des variables d'environnement nommées *ID*, *ARG* et *VAL*. La première ligne de la figure 3.4 est un exemple de paramétrage du mécanisme d'injection. Cet exemple correspond au code instrumenté de la figure 3.3 (colonne de droite). Ce premier exemple active le mécanisme d'injection identifié par la valeur $0x15$ (ligne 12 sur la figure 3.3) et injecte la valeur 1 dans la variable c (la cible numéro 0) plaçant ainsi le programme instrumenté dans un état interne erroné juste avant l'appel à la fonction *func* (ligne 15).

```
ID=0x00000015 ARG=0 VAL=1 ./mon_programme

ID=0x00000023 ARG=$RANDOM VAL=$RANDOM ./mon_programme
```

FIGURE 3.4 – Exemples de paramétrage des mécanismes d'injection

Notons que la désignation de la variable à corrompre est en fonction de l'emplacement de son adresse dans la liste des adresses de variables passées en paramètre de

la fonction d'injection. Dans l'exemple précédent nous injectons la variable *c*. Son adresse est la première dans la liste des cibles potentielles, c'est pourquoi elle est désignée par la valeur 0.

Toutefois, la fonction d'injection que nous utilisons considère que la valeur contenue par la variable d'environnement *ARG* est modulo le nombre total de variables dans la liste des cibles potentielles du point d'injection concerné. Cela permet de faciliter les campagnes d'injection. En effet, il est ainsi possible de choisir aléatoirement la variable à injecter sans avoir besoin de connaître préalablement le nombre de variables passées en paramètre de la fonction d'injection pour ce point précis du programme.

De même, toujours pour faciliter la réalisation des campagnes d'injections, la fonction d'injection que nous utilisons considère également que la nouvelle valeur de la variable ciblée et qui est contenue dans la variable d'environnement *VAL* est modulo la taille de l'emplacement mémoire utilisé pour contenir la valeur originale. À nouveau, ceci va nous permettre de choisir aléatoirement la nouvelle valeur de la variable après injection sans avoir besoin de connaître préalablement la taille de son emplacement mémoire.

Le second exemple de la figure 3.4 illustre cela en utilisant la variable d'environnement *RANDOM*, une variable d'environnement fournie par l'interpréteur de commande et qui renvoie à chaque lecture un nombre entier aléatoire. Dans cet exemple, c'est maintenant le mécanisme d'injection identifié par la valeur $0x23$ (ligne 19 sur la figure 3.3) qui est activé. Cependant, cette fois la variable qui va être la cible de l'injection est choisie aléatoirement parmi les variables *c*, *o* et *p* et la valeur injectée sera elle aussi choisie aléatoirement.

3.3 Résumé et discussion

Nous avons présenté dans ce chapitre une approche pour l'évaluation des mécanismes de détection hôtes. Plus précisément, nous proposons une méthode pour évaluer les capacités de détection d'un système de détection d'intrusion de niveau applicatif vis-à-vis des attaques contre les données de calcul. Pour cela, nous avons implémenté un mécanisme de simulation d'attaques qui repose sur l'injection de fautes en mémoire. Les mécanismes d'injection utilisent un modèle de faute qui permet de simuler pendant l'exécution du programme le résultat d'une attaque contre les données de calcul en allant directement modifier en mémoire les cibles potentielles pour un utilisateur malveillant.

Nous avons également présenté l'implémentation que nous avons réalisée afin d'utiliser notre modèle de faute pour évaluer le système de détection d'intrusion que nous avons présenté au chapitre précédent. Nous avons montré qu'une implémentation était réalisable à l'aide des mêmes outils d'analyse statique utilisés pour générer nos mécanismes de détection (néanmoins dans ce cas, uniquement la coupe

de programme est nécessaire). C'est pourquoi nous avons modifié notre outil pour qu'il génère maintenant aussi des mécanismes d'injection. Nous avons ainsi obtenu une implémentation du processus de génération des mécanismes d'injection que nous proposons et qui s'applique aux programmes écrits en langage C de manière entièrement automatisée.

Cependant, les mécanismes d'injection embarqués dans le programme restent neutres vis-à-vis de la campagne de tests (ils n'imposent que les cibles et les points d'injection). C'est à l'utilisateur de notre outil pour l'évaluation de choisir le nombre d'injections qui doivent être effectuées durant l'exécution du programme, le nombre de cibles pour chaque injection et la valeur à injecter pour chaque cible. Nous avons expliqué que l'objectif de nos campagnes de tests était de placer le système de détection d'intrusion à évaluer dans la situation qui lui est la moins favorable possible. Dans le cas du mécanisme de détection que nous proposons au chapitre précédent, nous avons montré pourquoi cela impliquait que les simulations d'attaque respectent nécessairement les caractéristiques suivantes :

- une seule injection doit être visée,
- une seul donnée doit être ciblée,
- la valeur injectée ne doit pas tenir compte de la valeur originalement contenue par la cible.

Nous avons aussi abordé la problématique de la propagation d'erreur et la problématique de l'impact. En effet, nous avons expliqué qu'une campagne de tests est nécessaire pour réellement évaluer notre approche pour la détection car le taux de détection réel est potentiellement supérieur au taux de détection théorique calculable aux différents points d'injection. Ceci est dû au fait que l'état erroné peut ne pas être détecté juste après l'injection mais que la détection peut avoir lieu plus loin dans l'exécution du programme. C'est ici qu'intervient la problématique de la propagation d'erreur.

Toutefois, la question de la pertinence de nos simulations d'attaque vis-à-vis des situations réelles d'attaque a également été posée. Cette fois c'est la problématique de l'impact qui a son importance ici. Nous avons expliqué que bien que le programme soit dans un état interne erroné, l'erreur peut être masquée par les instructions exécutées et l'injection n'aura donc ainsi pas d'impact sur le programme. Cependant, lorsque cet impact est observable et que notre mécanisme de détection d'intrusion n'a pas levé d'alerte, nous considérons qu'il s'agit là d'un faux négatif sûr. L'ensemble des injections où cette situation se produit permet alors de calculer un taux de faux négatifs correspondant au cas ou l'on est sûr que notre système de détection d'intrusion est en situation d'échec.

Afin de pouvoir observer cette éventuelle déviation comportementale, nous proposons d'analyser les traces d'exécution du programme au niveau de ses appels système mais aussi au niveau des données sorties par ce dernier. Par contre, nous avons également vu que bien qu'une erreur soit effectivement masquée, celle-ci peut pourtant parfaitement simuler le résultat d'une attaque bien réelle. Cela implique

alors que pour obtenir comme estimation une sur-approximation du taux de faux négatifs de notre approche, nous devons considérer tous les cas où l'injection n'a pas été détectée.

Au final, l'approche que nous proposons dans ce chapitre simule l'état erroné de la mémoire d'un programme ayant été la cible d'une attaque contre les données de calcul, que cette dernière est effectivement permis une intrusion ou non. Notre système de détection d'intrusion est justement conçu pour détecter ce type d'état erroné. Comme nous considérons que la détection d'une tentative d'intrusion, réussie ou non, est un argument en faveur de notre approche pour la détection, l'utilisation de la méthode d'évaluation est donc pertinente. Notre modèle de faute permet bien de simuler des attaques dans la situation la moins favorable pour notre système de détection d'intrusion. Mais nous n'avons pas moyen de savoir lorsque celles-ci sont pertinentes vis-à-vis des attaques réelles. Cependant, si on est capable de détecter une déviation comportementale suite à une injection, nous pouvons supposer que l'injection a ciblé les données du programme les plus susceptibles d'être intéressantes pour un utilisateur malveillant.

Chapitre 4

Protocole de tests et analyse des résultats

Au chapitre 2, nous avons présenté un système de détection d'intrusion capable de détecter au sein des programmes, durant leur exécution, des attaques contre les données de calcul. Au chapitre 3, nous avons présenté un mécanisme d'injection de fautes capable de simuler l'état erroné d'un programme suite à une attaque, durant son exécution, contre les données de calcul (on ne se soucie pas de la réussite ou non de l'attaque). Nous savons que notre approche pour la détection d'intrusion présente un taux de faux positifs nul. Afin d'évaluer de manière automatisée le taux de faux négatifs de notre approche, nous avons développé une plateforme de tests autour du mécanisme d'injection de fautes que nous proposons.

L'objectif de cette plateforme d'évaluation est de conduire une campagne d'injections sur une application donnée et de collecter les informations nécessaires à l'estimation du taux de faux négatifs de notre approche pour la détection. Bien sûr, l'application à tester doit d'abord être instrumentée pour l'injection de fautes selon le modèle que nous avons présenté dans la section 3.1.1.4. Nous voulons aussi dans une certaine mesure évaluer la pertinence de nos simulations d'attaques vis-à-vis des attaques réelles et ceci dans le but d'affiner l'évaluation du taux de faux négatifs du mécanisme de détection considéré.

Pour cela, nous devons être capable de détecter une déviation comportementale du programme en cours de test. En effet, nous allons considérer que les simulations d'attaques qui ont effectivement provoqué une déviation comportementale du programme font partie des plus pertinentes vis-à-vis des attaques réelles contre les données de calcul (on est sûr que ces injections mettent le programme dans un état de défaillance). On va donc également considérer que l'ensemble de ces injections qui ne sont pas détectées par notre système de détection d'intrusion constitue un ensem-

ble de faux négatifs plus grave. Toutefois, cette information n'est qu'un complément de la mesure de l'ensemble des faux négatifs. Il n'est en effet pas possible de garantir que toutes les injections qui n'ont pas provoqué de déviation comportementale sont effectivement bénignes.

Dans ce chapitre, nous présentons deux choses. Tout d'abord, nous détaillons le fonctionnement de la plateforme de tests que nous avons développée. Pour cela, nous abordons la question de l'écriture des scénarios d'exécution. L'objectif ici est d'obtenir une bonne couverture de code afin qu'une campagne de tests puisse cibler un maximum de cibles potentielles. Nous abordons également la question de la collecte des informations. L'objectif cette fois est d'obtenir les données nécessaires à l'analyse des résultats. Ensuite, nous présentons les résultats obtenus avec plusieurs programmes. Dans le cadre des campagnes d'injections, il s'agit uniquement d'applications réseau. Nous présentons les résultats de l'analyse statique (nombre d'invariants calculés et durée de l'analyse), la surcharge à l'exécution des mécanismes de détection (à l'aide d'un logiciel de référence) ainsi que les résultats des campagnes d'injections (avec et sans déviation comportementale).

4.1 La plateforme de tests

Au centre de notre plateforme de tests, on trouve un moniteur d'exécution. En effet, bien que l'injection soit réalisée par le mécanisme que nous avons présenté au chapitre 3, pour effectuer une campagne de tests, ce mécanisme doit être piloté de l'extérieur du processus en cours d'exécution. De plus, pour chaque test, la plateforme doit pouvoir se synchroniser avec le processus ciblé par le mécanisme d'injection. C'est le rôle du moniteur d'exécution de piloter le mécanisme d'injection et de se synchroniser avec le programme en cours de test. Notons que par synchronisation nous entendons deux choses : l'exécution du processus selon un scénario bien précis et la collecte d'informations. Toute la plateforme de tests s'articule donc autour de ce moniteur d'exécution et celui-ci joue plusieurs rôles. Nous allons maintenant aborder en détail l'ensemble de ces rôles.

Tout d'abord, c'est le moniteur d'exécution qui est responsable des multiples exécutions du programme en cours de test. Les exécutions du programme se font selon un ensemble de scénarios prédéterminés qui doivent être fournis au moniteur d'exécution. Nous abordons la question de la génération des scénarios mais aussi la problématique de la couverture de code de ces scénarios dans la section 4.1.1. Toutefois, une étape préalable à la campagne de tests et qui implique les scénarios doit tout d'abord être réalisée.

En effet, quand nous avons développé le moniteur, nous avons dû faire un choix quant à la répartition des injections. Plus précisément, nous avons choisi de répartir les injections de manière égale entre toutes les variables de tous les points d'injections atteignables par le jeu de scénarios considéré. De cette manière, une campagne de

tests donnée cible de manière équitable l'ensemble des cibles potentielles pour un utilisateur malveillant.

Pour obtenir cela, le moniteur d'exécution va préalablement exécuter chacun des scénarios sans réaliser d'injection. En effet, comme nous l'avons vu à la section 3.2.2.1, le code d'injection que nous avons fourni à l'instrumenteur de code signale sa présence (et le numéro de l'injection) à chaque fois que l'exécution passe par un point d'injection. Cette étape préalable permet donc au moniteur d'exécution de connaître pour chacun des scénarios l'ensemble des points d'injection atteignables par le programme. Le moniteur d'exécution n'a plus qu'à en déduire les ensembles de scénarios qui permettent d'atteindre chaque injection pour pouvoir réaliser une campagne de tests avec la répartition des injections que nous avons énoncée dans le paragraphe précédent. Notons que c'est également pendant cette phase préalable que le moniteur d'exécution enregistre en situation de fonctionnement normal les résultats et les traces d'exécution de référence pour chacun des scénarios de tests. Ces informations seront exploitées durant la phase d'analyse des résultats (voir section 4.2.2.1).

Ensuite, le moniteur d'exécution est également responsable de l'activation du mécanisme d'injection au point considéré. Ceci est réalisé en utilisant des variables d'environnement comme nous l'avons expliqué dans la section 3.2.2.1. Une seule injection est activée lors de l'exécution du programme en cours de test et celle-ci ne cible qu'une seule variable à la fois. Nous avons justifié ces choix dans les sections 3.1.2.1 et 3.1.2.2. Le numéro de l'injection et celui de la variable à injecter sont choisis pour obtenir la répartition que avons énoncée précédemment. La valeur à injecter, quant à elle, est choisie de manière aléatoire. Nous avons justifié ce dernier choix dans la section 3.1.2.3.

La valeur à injecter que va fournir le moniteur d'exécution est choisie aléatoirement sur une plage de valeurs équivalente à celle d'un entier d'une taille de 64 bits. Ceci correspond à la taille maximum des entiers pouvant être ciblés par le mécanisme d'injection. Mais cela ne pose pas de problème même quand la cible est un entier de taille inférieure. En effet, nous avons vu dans la section 3.2.2.1 que le code d'injection que nous avons fourni à l'instrumenteur de code et sur lequel repose le mécanisme d'injection n'utilise que les bits nécessaires en fonction de la taille de la variable ciblée par l'injection. Par exemple, si la cible de l'injection est un entier d'une taille de 32 bits, alors seuls les 32 bits de poids faibles de la valeur aléatoire fournie par le moniteur d'exécution seront utilisés pour réaliser la corruption de la variable ciblée.

Enfin, le moniteur d'exécution se charge de la collecte des informations qui sont nécessaires à l'estimation du taux de faux négatifs du système de détection d'intrusion en cours de test. Plus précisément, en plus des messages et des éventuelles alertes envoyés par le système de détection d'intrusion, celui-ci sauvegarde l'état du processus à la fin de l'exécution du scénario choisi (voir section 4.1.2.2). De plus, il conserve également une trace d'exécution du programme depuis le début de l'exécution du

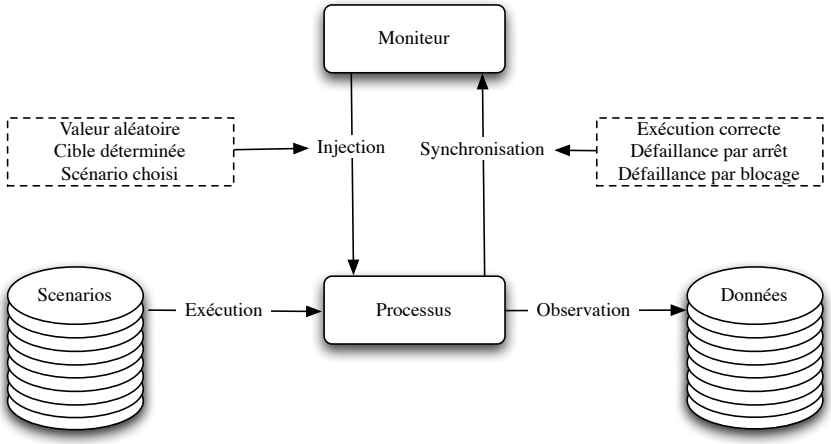


FIGURE 4.1 – Plateforme de tests

scénario. La figure 4.1 illustre sur un schéma l'ensemble de la plateforme de tests. Nous allons maintenant détailler dans la suite de cette section les problématiques d'écriture des scénarios et de collecte des informations.

4.1.1 Scénarios d'exécution

Les injections qui vont être réalisées par notre plateforme de tests simulent le résultat d'une attaque contre les données de calcul qui ne réalise qu'une seule corruption de données et qui ne cible qu'une seule variable. Nous avons justifié ces choix dans les section 3.1.2.1 et 3.1.2.2. En situation réelle, lorsqu'une attaque de ce type a lieu, celle-ci peut être caractérisée par la variable qu'elle cible, le moment où la corruption est réalisée et le chemin d'exécution qui l'a menée jusqu'au point de corruption. Nous parlerons de configuration de l'attaque pour désigner ces trois caractéristiques à propos d'une attaque donnée.

Cependant, lors d'une campagne de tests, parmi les variables dont dépendent les appels de fonction dans le programme, nous ne faisons aucune supposition sur celles qui ont la capacité, lorsqu'elles sont corrompues, de faire dévier le comportement du programme de manière dangereuse (et qui sont donc susceptibles d'être ciblées par un utilisateur malveillant). Nous ne faisons également aucune supposition sur l'emplacement dans le programme des éventuelles vulnérabilités (et donc sur le moment où la corruption d'une variable peut avoir lieu). Enfin, nous ne faisons aussi aucune supposition sur l'état du programme au moment de l'attaque (et donc sur les chemins d'exécution qui peuvent mener à la corruption d'une variable). Idéalement, afin de ne rater aucune configuration d'attaque potentiellement représentative d'une attaque réelle, il nous faudrait procéder à au moins une injection pour chaque configuration d'attaque possible.

Dans le cadre d'injections réalisées à l'aide du mécanisme que nous avons présenté au chapitre 3, nous avons choisi de nous limiter à un parcours exhaustif de seulement deux caractéristiques des configurations d'attaque : la variable ciblée et le point de corruption. Pour cela, nous devons être capable de réaliser une injection pour l'ensemble des variables de tous les points d'injection résultant de l'instrumentation du programme. Nous verrons dans la suite de cette section que ceci n'est pas toujours possible et nous illustrerons cela par plusieurs exemples.

En effet, si on peut atteindre un point d'injection particulier, alors cela ne présente aucune difficulté de réaliser une injection pour chacune des variables qui le concerne. Mais certains points d'injection peuvent ne pas être évidents à atteindre. Le jeu de scénarios fourni au moniteur d'exécution doit donc avoir comme objectif de permettre d'atteindre le plus de points d'injection possibles. Pour cela, nous définissons donc un scénario comme l'ensemble des informations ou l'enchaînement d'entrées qui permettent de prévoir de manière déterministe le comportement du programme afin de s'assurer que son exécution passe bien par un certain nombre de points d'injection. Cela comprend bien sûr des paramètres d'exécution du programme (ses arguments et ses éventuels fichiers de configuration) et les données à lui fournir en entrée, mais également si besoin l'état de l'environnement au moment de son exécution. Ces scénarios sont analogues aux scénario de tests utilisés en test logiciel. Notons que si plusieurs scénarios permettent d'atteindre un même point d'injection, chacun d'entre eux sera exécuté au moins une fois pour chaque variable qui peut être corrompue par ce point d'injection particulier. Ceci ne garantit pas un parcours exhaustif de la dernière caractéristique, le chemin d'exécution qui a mené au point de corruption, mais cela augmente les chances d'en tester un plus grand nombre.

Nous avons donc choisi de mesurer la couverture des scénarios en fonction du nombre total d'injections que ces derniers permettent d'atteindre. Pour cela, le moniteur commence par exécuter chaque scénario sans spécifier d'identifiant d'injection. Chacun des appels au code d'injection va alors signaler sa présence durant l'exécution du scénario. C'est ce qui permet au moniteur d'exécution de connaître le taux de couverture de l'ensemble des scénarios. C'est aussi ce qui permet au moniteur d'exécution de connaître l'ensemble des injections atteignables par chaque scénario. Cette dernière information est ensuite utilisée pour créer pour chaque point d'injection atteignable la liste des scénarios qui permettent de l'atteindre. Avant de lancer une campagne de tests, nous nous fixons comme objectif d'avoir un jeu de scénarios ayant un taux de couverture des mécanismes d'injection suffisant dans la pratique (supérieur ou égal à 80%).

Ce sont ces listes qui permettent ensuite au moniteur d'exécution de répartir durant la campagne de tests les injections de manière égale entre chaque variable de chaque point d'injection, et ce en s'assurant que chaque scénario qui permet d'atteindre un point d'injection donné est au moins exécuté une fois pour chacune des variables concernées. Notons que c'est aussi à ce moment que le moniteur enregistre

les traces d'exécution des scénarios en situation de fonctionnement normal. Ces dernières sont nécessaires durant la phase d'analyse des résultats pour déterminer le niveau de sévérité des injections qui ne sont pas détectées par notre mécanisme de détection d'intrusion. Cette dernière problématique est abordée plus en détails dans la section 4.2.2.1.

Dans les campagnes de tests que nous avons menées, nous avons essentiellement travaillé avec des applications qui fonctionnent en réseau et plus précisément qui reposent sur une paradigme client-serveur. Dans cette situation, les scénarios de tests comprennent chacun un scénario d'exécution coté serveur et un scénario d'exécution coté client. Notons que dans nos campagnes de tests, c'est toujours le serveur qui est le programme instrumenté. Cela implique qu'un scénario de tests peut ne pas nécessiter de scénario côté client. C'est par exemple le cas des scénarios qui permettent d'activer des injections qui ne sont atteintes par le programme que si le serveur ne réussit pas à démarrer correctement. Si le programme instrumenté était le client, alors la même situation pourrait se produire avec ce dernier. Nous allons maintenant détailler la problématique d'écriture des scénarios côté serveur, puis nous ferons de même pour la problématique côté client et enfin nous parlerons des cas où il est difficile d'écrire un scénario pour atteindre certains points d'injection et que nous avons choisis de ne pas traiter.

4.1.1.1 Scénario d'exécution du serveur

La première chose à considérer lors de l'écriture des scénarios côté serveur, c'est l'environnement dans lequel va s'exécuter le programme, et ce pour deux raisons. Tout d'abord parce que l'environnement peut influencer le chemin d'exécution du processus. Par exemple, un serveur peut chercher son fichier de configuration dans le répertoire */etc*. S'il ne le trouve pas, il charge alors une configuration par défaut. Si le code responsable de la configuration par défaut contient des mécanismes d'injection, alors la présence d'un fichier de configuration dans le répertoire */etc* va empêcher d'atteindre ces points d'injection. Pour résoudre ce problème, le moniteur d'exécution devra préalablement modifier l'environnement du programme, c'est-à-dire dans le cas de notre exemple, modifier le système de fichiers.

L'autre raison pour laquelle il est important de considérer l'environnement d'exécution du processus, c'est que le programme instrumenté peut lui-même modifier son environnement. Prenons comme exemple un programme qui, entre autres, enregistre des informations dans un fichier journal. Au début de son exécution, le programme regarde si le fichier existe. Si tel est le cas, il peut donc directement commencer à ajouter des informations à la fin du fichier. Dans le cas contraire, il doit alors d'abord le créer. Si le code responsable de la création du fichier contient un mécanisme d'injection, alors celui-ci ne sera atteignable qu'à la première exécution du programme. Toutes les exécutions suivantes ne pourront pas atteindre ce point d'injection. Ceci pose bien sûr problème pour la campagne de tests, le programme étant exécuté de

nombreuses fois. Au final, pour contourner le problème, le moniteur d'exécution doit remettre l'environnement dans son état initial à la fin de chaque test, c'est-à-dire dans le cas qui nous intéresse, supprimer le fichier journal (ou le déplacer si celui-ci fait partie des informations à collecter, voir la section 4.1.2.1).

Il reste encore à prendre en compte l'ensemble des paramétrages possibles dans l'écriture des scénarios côté serveur. En effet, la plupart des programmes possèdent de nombreuses options paramétrables via la ligne de commande ou via des fichiers de configuration. Chaque paramètre peut influencer le comportement du serveur et donc permettre d'atteindre certains points d'injection. C'est le cas du précédent exemple à propos du protocole *SSH1*. De manière générale, cette tâche est fastidieuse car elle nécessite de se plonger dans le code du serveur. Il est par exemple possible de tomber sur un mécanisme d'injection qui ne peut être atteint que grâce à une combinaison bien précise de plusieurs options. Il est toutefois possible d'avoir recours à des outils d'analyse statique pour assister dans cette tâche, notamment les outils capables de générer le graphe des appels de fonction.

4.1.1.2 Scénario d'exécution du client

Dans la section précédente, nous avons abordé la problématique de l'écriture des scénarios d'exécution coté serveur. Il faut savoir que les paramètres d'exécution du serveur peuvent influencer l'écriture des scénarios coté client. Prenons comme exemple un serveur *web*. Le scénario cherche, côté client, à exécuter un script *cgi* stocké sur le serveur, par exemple en demandant la page *http ://mon_ serveur/cgi-file/hello.cgi* via le programme *telnet*. Cependant, le chemin d'accès au répertoire contenant les scripts *cgi* est une information configurable côté serveur. Si on venait par exemple à remplacer la valeur par défaut *cgi-file* par *cgi-script*, alors les scénarios d'exécution côté client devront être modifiés en conséquence.

Une des difficultés d'écriture des scénarios d'exécution côté client réside dans le fait qu'il faut arriver à couvrir le plus possible (sinon l'intégralité) du protocole réseau. En effet, si on décrit le protocole utilisé par le serveur sous forme d'un graphe, alors chacune des arêtes de ce graphe représente côté serveur des blocs de code susceptibles de contenir des mécanismes d'injection. L'ensemble des scénarios doivent donc permettre de couvrir l'ensemble des arêtes de ce graphe. Notons qu'à nouveau ceci peut être influencé par la configuration du serveur. Prenons comme exemple un serveur *SSH* dont la configuration par défaut autorise seulement pour se connecter le protocole *SSH2*. Pourtant, le code en charge du protocole *SSH1* va tout de même être analysé par notre outil. Celui-ci est donc également susceptible de contenir, après instrumentation, des mécanismes d'injection. Même si coté client on force l'utilisation du protocole *SSH1*, le scénario d'exécution devra tout de même modifier la configuration du serveur pour autoriser explicitement ce protocole.

Une autre difficulté dans l'écriture des scénarios d'exécution côté client réside dans le fait que couvrir tout le protocole réseau n'est pas suffisant. En effet, un

serveur bien écrit (ce qui n'exclut pas qu'il puisse tout de même contenir des vulnérabilités) doit s'assurer que le client respecte bien le protocole et dans le cas contraire gérer les erreurs. À nouveau, ces vérifications et la gestion respective des erreurs sont autant de code susceptible de contenir des mécanismes d'injection. Les scénarios d'exécution doivent donc aussi, côté client, chercher à envoyer des jeux de données erronées afin de pouvoir atteindre ces points d'injection. Bien souvent, ceci n'est pas possible avec un client standard. Le scénario doit donc dans ce cas gérer lui-même l'envoi des données au serveur. Notons qu'on se rapproche ici de la problématique du *fuzzing* (voir la section 1.4).

4.1.1.3 Cas problématiques

En écrivant les scénarios, aussi bien pour le serveur que pour le client, nous avons remarqué que certains points d'injection étaient très difficiles voir impossibles à atteindre durant l'exécution du programme. Nous avons alors identifié deux catégories d'injections qui posent ce problème d'atteignabilité. Tout d'abord, celui-ci se pose pour les mécanismes d'injection qui se trouvent dans des blocs d'instructions dédiés au traitement des erreurs liées à l'environnement d'exécution. La figure 4.2 illustre cela dans le cadre d'une tâche prise en charge par un fil d'exécution dédié. Le code original est situé dans la colonne de gauche. On peut voir que le programmeur a pris soin de vérifier que la création à la ligne 02 d'un nouveau fil d'exécution s'est bien déroulée correctement. Cette vérification a lieu à la ligne 07 (la valeur retournée par l'appel de fonction ne doit pas valoir -1). En cas d'erreur, le programme fait appel à la fonction *exit_error* qui arrête proprement l'exécution du programme après avoir enregistré l'erreur dans le fichier journal.

```
00. int ret;
01.
02. ret = pthread_create(handle
03.                      &thr_attr,
04.                      htloop,
05.                      NULL);
06.
07. if (ret == -1){
08.
09.
10.
11.   exit_error("pthread_create");
12. }
```

```
00. int ret;
01.
02. ret = pthread_create(handle
03.                      &thr_attr,
04.                      htloop,
05.                      (void *)0);
06.
07. if (ret == -1){
08.
09.   inject(0x0000000b, 1,
10.          &ret, sizeof(int));
11.   exit_error("pthread_create");
12. }
```

FIGURE 4.2 – Exemple d'erreur difficile à reproduire

Ce même code, une fois instrumenté par notre outil, est situé dans la colonne de droite. On peut voir qu'un mécanisme d'injection a été placé à la ligne 09 juste

avant l'appel à la fonction *exit_ error*. En effet, la valeur -1 fait partie des valeurs que peut retourner un appel à la fonction *pthread_ create*. L'analyse de valeur a donc considéré que ce point particulier du programme n'est pas du code mort et qu'il est parfaitement atteignable durant l'exécution du processus. De plus, on trouve parmi les dépendances de cet appel la variable entière *ret*, dont la valeur participe aux décisions qui permettent d'atteindre ce point particulier du programme. On peut donc bien ajouter un mécanisme d'injection à cet endroit.

Cependant, cette injection est particulièrement difficile à atteindre. En effet, pour y arriver, le scénario de tests doit pouvoir mettre le système dans un état qui ferait échouer la création du nouveau fil d'exécution (par exemple en saturant la mémoire de la machine afin de faire échouer les allocations mémoire effectuées par la fonction *pthread_ create*). De plus, cet état ne doit être atteint que pour cet appel de fonction précis. L'atteindre plus tôt c'est prendre le risque que l'exécution du programme échoue avant d'avoir atteint le point d'injection désiré. Une première solution serait de modifier l'analyse de valeur pour qu'elle suppose que la fonction *pthread_ create* n'échoue jamais. Si cette solution marche pour l'exemple de la figure 4.2, elle n'est pas satisfaisante dans le cas général.

En effet, cela suppose d'une part que tout code de gestion d'erreur suite à un appel de fonction dont on décide qu'il ne peut pas échouer sera considéré comme du code mort. Pourtant, on peut envisager qu'une injection fasse échouer un de ces appels. Supposons par exemple que le cas de la figure 4.2 s'inscrivent dans une fonction de plus grande taille. Si dans ce cas un mécanisme d'injection en amont dans le code venait à corrompre l'un des paramètres de l'appel à la fonction *pthread_ create* ligne 02, ce dernier pourrait échouer. Si l'injection n'est pas détectée immédiatement, on peut envisager qu'une assertion située dans le code de gestion d'erreur la détecte par propagation de l'erreur. Mais si ce code est considéré comme mort par l'analyse de valeur, alors aucun ensemble de variation ne sera calculé et donc il ne pourra pas y avoir d'assertion.

D'autre part, cette solution n'est envisageable dans le cas de la figure 4.2 que parce que la fonction concernée, *pthread_ create*, est une fonction standard du système. En effet, pour ce type d'appel de fonction, l'analyse de valeur a connaissance de l'ensemble de valeurs qui peut être retourné grâce à l'écriture d'un *stub* correspondant pour chaque fonction (voir section 2.4.2.1). Il suffit alors de modifier en conséquence le *stub* des fonctions concernées pour que l'analyse de valeur considère que ces dernières n'échouent jamais. Cependant, cette situation peut également se produire pour des fonctions dont le code est fourni à l'analyse de valeur, notamment à cause des sur-approximations effectuées par l'interprétation abstraite. Cette solution n'est donc bien pas envisageable dans le cas général, ce qui la rend inutilisable dans le cadre d'une approche automatisée.

Notons toutefois que dans le cas d'un code de gestion d'erreur suite à un appel à une fonction standard, nous avons envisagé l'utilisation d'une bibliothèque de fonctions personnalisée. Celle-ci serait chargée à l'exécution via le mécanisme de

LD_PRELOAD et aurait pour but d'intercepter les appels de fonction qui nous intéressent afin de les faire échouer à un moment choisi par le moniteur d'exécution. Nous avons cependant choisi de ne pas complexifier l'écriture de nos scénarios avec un tel mécanisme. En effet, ce type d'appel à la fonction d'injection représente un très faible nombre du total des points d'injection. Nous avons d'ailleurs pu atteindre dans tous les cas des taux de couverture des injections supérieurs à 80% sans avoir recours à ce stratagème.

Abordons maintenant l'autre type d'injections que nous avons identifié comme posant un problème d'atteignabilité. Il s'agit des mécanismes d'injection qui se trouvent dans des blocs d'instructions qui ne seront jamais exécutés à cause d'une faute de conception. La figure 4.3 illustre cela par un exemple extrait du serveur *web ihttpd* [IHTTP]. Ce serveur *web* n'implémente que les requêtes de type *GET* et *POST*. D'après la norme *HTTP* 1.0, il est censé renvoyer une erreur 501 si le type de requête demandée n'est pas implémentée. Dans le code original qui se trouve en haut dans la colonne de gauche, on peut voir que le développeur s'est trompé dans la vérification de cette condition. Une même chaîne de caractères ne pouvant à la fois être égale à "GET" et "POST", le code qui retourne la valeur 501 ne sera donc jamais atteint.

```
00. if (strcasecmp(mt, "GET") == 0
01. && strcasecmp(mt, "POST") == 0){
02.
03.   logErr("%s unsupported", mt);
04.   return 501; /*unimplemented*/
05. }
```

```
00. if (strcasecmp(mt, "GET") != 0
01.  && strcasecmp(mt, "POST") != 0){
02.
03.   logErr("%s unsupported", mt);
04.   return 501; /*unimplemented*/
05. }
```

```
00. int tmp_0;
01. int tmp_1;
02.
03. no_inject();
04. tmp_0 = strcasecmp(mt, "GET");
05.
06. if (tmp_0 == 0){
07.
08.  inject(0x00000c07, 1,
09.         &(tmp_0), sizeof(int));
10.  tmp_1 = strcmp(mt, "POST");
11.
12.  if (tmp_1 == 0){
13.
14.    inject(0x00000c0a, 2,
15.           &(tmp_0), sizeof(int),
16.           &(tmp_1), sizeof(int));
17.    logErr("%s unsupported", mt);
18.
19.    __retres = 501;
20.    goto return_label;
21.  }
22. }
```

FIGURE 4.3 – Exemple de code impossible à atteindre

Ceci n'empêche nullement le serveur *web* de fonctionner correctement. En effet, d'autres vérifications, qui se trouvent plus loin dans le code du programme, compensent cette faute de conception. Cependant, l'analyse de valeur, même en lui fournissant une implémentation de la fonction *strcmp* et non un *stub*, n'est pas capable de détecter que ce bloc de code est une portion de code mort. Le code, une fois instrumenté par notre outil, se trouve dans la colonne de droite de la figure 4.3. On peut voir qu'une injection a pu être ajoutée juste avant l'appel à la fonction *logErr* à la ligne 17 (dans le code original, cet appel se trouve à la ligne 03). Cette injection se trouvant dans la portion de code mort, il n'est pas possible pour un scénario d'exécution de l'atteindre.

Pour contourner le problème, nous pourrions corriger cette faute en modifiant le code source du programme de manière à obtenir ce que l'on voit en bas de la colonne de gauche de la figure 4.3. Toutefois, rappelons que le but de nos campagnes de tests est d'évaluer un système de détection d'intrusion qui fonctionne au niveau applicatif et ce malgré les vulnérabilités présentes dans les programmes. Nous faisons donc le choix, dans ce type de cas, de ne pas corriger les fautes de conception afin que le contexte d'évaluation de notre approche pour la détection soit la plus proche possible d'une situation réelle.

4.1.2 Informations collectées

Afin d'évaluer la couverture de détection du système de détection d'intrusion que nous proposons, un certain nombre d'informations doivent être collectées durant la phase de tests. Ces informations permettront notamment d'estimer le taux de faux négatif mais aussi de déterminer les injections qui sont les plus représentatives d'une éventuelle attaque. Pour chaque exécution d'un scénario de tests, le moniteur d'exécution va bien sûr enregistrer l'ensemble des messages émis par les mécanismes de détection et d'injection que nous avons ajoutés au programme.

Mais des informations supplémentaires doivent également être collectées. Tout d'abord, le moniteur d'exécution va également enregistrer les traces d'exécution du serveur. Il s'agit du résultat de l'exécution mais aussi de l'enchaînement des appels système. Enfin, il va aussi enregistrer l'état d'arrêt du scénario. Nous allons maintenant détailler dans la suite de cette section pourquoi nous avons besoin de ces informations supplémentaires et comment le moniteur d'exécution y accède.

4.1.2.1 Les traces d'exécutions

Il s'agit ici de collecter les informations qui permettent de déterminer si le comportement du programme a dévié ou non suite à l'injection. Ces informations sont utilisées dans la section 4.2.2.2 pour déterminer quelles injections présentent le plus de risque afin d'affiner l'évaluation (voir la section 4.2.2.1 pour la méthode). La première chose que l'on collecte, ce sont les résultats produits par le programme. C'est notamment le cas des informations envoyées sur les sorties erreur et standard.

Mais ceci peut varier d'un programme à l'autre. Par exemple, on peut également juger utile de récupérer le contenu d'un fichier journal (qu'il faut donc remettre à zéro avant le prochain test). À noter que dans le cas des sorties erreur et standard, il est parfois utile de collecter les résultats produits par le client. En effet, passé un certain point dans l'exécution du programme, certains serveurs redirigent leurs sorties erreur et standard à destination du client.

La seconde chose que nous avons besoin de collecter, c'est l'enchaînement des appels système effectués par le processus durant toute la durée du scénario. À cela nous ajoutons la valeur des arguments de chacun des appels système. Ceci nous permettra de détecter une déviation comportementale quand bien même les résultats produits par le programme seraient inchangés. Pour collecter ces informations, nous avons choisi d'utiliser *Strace* [STRAC]. Il s'agit d'un outil de déboguage pour *Linux* qui permet de surveiller les appels système effectués par un programme donné mais aussi de connaître tous les signaux qu'il reçoit. Pour l'utiliser au sein de notre plateforme de tests, le moniteur d'exécution va tout simplement se reposer sur lui pour lancer la commande associée au programme instrumenté dans les scénarios de tests.

4.1.2.2 L'état d'arrêt du scénario

Une injection peut avoir d'autres incidences sur le processus qu'elle cible que de faire dévier le comportement du programme. Cela peut aller jusqu'à bloquer l'exécution du processus voir l'interrompre inopinément. Bien sûr, de tels évènements sont normalement détectables au niveau des traces d'exécution. Toutefois, nous voulons pouvoir les distinguer des autres déviations comportementales. En effet, nous considérons qu'une injection qui a réussi à faire dévier le comportement du programme sans pour autant mettre en péril la viabilité du processus est plus représentative de ce que cherche à réaliser un utilisateur malveillant.

Pour cette raison, le moniteur d'exécution va chercher à déterminer l'état d'arrêt du scénario avant d'ajouter celui-ci aux informations enregistrées pour le test qui vient de se dérouler. La plateforme de tests que nous avons développée considère trois états possibles : le scénario s'est déroulé jusqu'à son terme, le processus s'est interrompu de manière inattendue et enfin ce dernier s'est trouvé dans une situation de blocage. Lorsque le dernier cas se présente, le moniteur d'exécution doit lui-même forcer l'arrêt du processus pour terminer le test en cours. Si le scénario du test en cours d'exécution ne permet pas de déterminer quand cette situation se produit, alors cet arrêt forcé doit avoir lieu après une certaine limite de temps. Cette limite de temps doit avoir été choisie suffisamment grande afin de pouvoir supposer que le programme est effectivement bloqué.

Notons également que si le processus s'interrompt de manière inattendue ou bien si celui-ci est arrêté de force, alors un délai supplémentaire doit être respecté avant de pouvoir passer aux tests suivant. En effet, lorsqu'un processus serveur se termine de la sorte, il n'est pas possible de faire immédiatement à nouveau appel à la fonction

réseau *bind*. Sans ce délai supplémentaire, il y a un risque que plusieurs des scénarios de tests à suivre puissent échouer avant d'avoir pu réaliser l'injection.

4.2 Résultat de l'évaluation

Nous allons maintenant présenter les résultats de l'évaluation du système de détection d'intrusion que nous avons présenté au chapitre 2. Pour cela, nous commençons par aborder les performances liées à l'instrumentation. D'une part, nous présentons l'évaluation des performances du processus d'instrumentation en terme de durée d'analyse et d'invariants générés. D'autre part, nous présentons l'évaluation des performances de l'instrumentation elle-même en terme de surcharge à l'exécution. Nous abordons ensuite les performances liées à la détection. C'est dans cette section que nous donnons les résultats des campagnes de tests que nous avons réalisées à l'aide de la plateforme présentée dans ce chapitre. D'une part, nous expliquons comment nous caractérisons la sévérité des simulations d'attaques à l'aide d'un autre mécanisme de détection. D'autre part, nous présentons une analyse des résultats des campagnes de tests.

4.2.1 Évaluation de l'instrumentation

Dans cette section, nous allons nous intéresser à deux aspects de notre instrumentation : la performance du processus d'instrumentation et la performance des mécanismes de détection ajoutés par l'instrumentation. L'évaluation du processus d'instrumentation consiste à évaluer en fonction des programmes la durée d'analyse nécessaire au calcul des invariants mais aussi le nombre d'invariants ainsi générés. Nous pourrons alors voir si ces performances varient d'un programme à l'autre et s'il est possible d'identifier l'origine de ces différences.

L'évaluation des mécanismes de détection eux-même a pour objectif de répondre à la question suivante : quelle surcharge à l'exécution est induite par notre instrumentation. Nous nous attendons à ce que cette surcharge soit relativement faible. En effet, très peu de nouvelles instructions sont finalement ajoutées au code source original. Cependant, nous allons vérifier cela par un ensemble de mesures effectué à l'aide d'un outil dédié à cette tâche.

4.2.1.1 Performance de l'analyse

Nous avons présenté dans la section 2.3.1 deux exemples d'attaques contre les données de calcul exploitant une vulnérabilité réelle et permettant de réaliser une intrusion sans déviation détectable du flot d'exécution du programme. Nous avons appliqué notre mécanisme d'instrumentation sur la portion de code vulnérable et nous avons obtenu les invariants capables de détecter ces attaques (voir la colonne

de droite de la figure 2.3). Toutefois, pour évaluer pleinement notre système de détection d'intrusion, il convient de tester notre approche sur des programmes entiers, sans connaissance a priori des vulnérabilités. Pour cela, nous choisissons de cibler pour notre évaluation des programmes réels mais néanmoins de taille modeste afin de limiter le temps nécessaire à l'instrumentation mais aussi à l'écriture des scénarios d'exécution (nous avons notamment remplacé *OpenSSH* [OSSH] par *Dropbear SSH* [DROPB]).

	lbm	mcf	libquantum	bzip2	milc	sjeng
Nombre de lignes de code	1267	2077	3567	7292	12837	13291
Nombre d'appels de fonction	72	88	228	134	1274	1718
Durée d'analyse en minutes	12	8	122	1044	4123	5245
Nombre d'invariants générés	28	46	114	96	293	729
Taux d'instrumentation	38%	52%	50%	72%	23%	42%

TABLE 4.1 – Résultat de l'instrumentation des applications locales

Nous avons utilisé notre outil sur deux types de programmes. Des applications locales et des applications réseaux. Les applications locales sont celles que nous utilisons dans la section 4.2.1.2 pour évaluer la surcharge à l'exécution de nos mécanismes de détection. Les applications réseaux sont celles que nous utilisons à la section 4.2.2 pour évaluer le taux de couverture de nos mécanismes de détection. Nous utilisons l'ensemble de ces programmes pour évaluer la performance de l'analyse. Les résultats de l'évaluation des performances de l'analyse sont présentés dans le tableau 4.1 pour les applications locales et dans le tableau 4.2 pour les applications réseaux.

	dropbear	ssmtp	fnord	ihttpd	nullhttpd
Nombre de lignes de code	11177	2717	2622	1180	5968
Nombre d'appels de fonction	429	314	232	289	399
Durée d'analyse en minutes	962	19	224	1	317
Nombre d'invariants générés	257	237	6	109	234
Taux d'instrumentation	60%	75%	3%	38%	58%
Nombre d'injections générés	371	261	209	232	356
Taux d'instrumentation	86%	83%	90%	80%	89%
Nombre de cibles potentielles	1194	1872	5378	1248	2082
Nombre de cibles vérifiées	439	1829	40	363	1169
Couverture des invariants	36,8%	97,7%	0,7%	29,1%	56,1%

TABLE 4.2 – Résultat de l'instrumentation des applications réseaux

Tout d'abord, pour comparer les résultats en terme de durée d'analyse, il nous faut connaître la taille des programmes analysés. En effet, plus le programme à analyser est de taille conséquente, plus grande sont les chances que la durée d'analyse soit longue. Pour comparer la taille des programmes, nous avons choisi de nous fonder sur le nombre de lignes dans leur code source. Toutefois, afin de limiter le biais dû au style de programmation des développeurs respectifs des applications concernées, les codes source ont préalablement été modifiés. En premier lieu, nous avons supprimé tous les commentaires. Les lignes de commentaires augmentent la taille du programme mais pas la durée d'analyse. Puis, nous avons réindenté automatiquement les codes source. Ceci nous permet de normaliser le rapport entre le nombre de lignes des codes source et le nombre d'instructions qu'ils contiennent. Ensuite, pour analyser les résultats en terme d'invariants générés, nous allons tout simplement comparer les taux d'instrumentation des programmes. Ces taux sont calculés par rapport à l'ensemble des appels de fonctions présents dans le code source des programmes.

Malgré l'influence du nombre d'instructions sur la durée de l'analyse, on peut voir que cette durée peut grandement varier entre deux programmes de taille comparable. Par exemple, *fnord* [FNORD] est un programme presque douze fois plus long à analyser que *ssmtp* [SSMTP], bien que ces deux programmes soient de taille comparable (respectivement 2622 lignes et 2717 lignes). Pour comprendre cela, ce n'est donc pas seulement le nombre d'instructions dont il faut tenir compte mais aussi les types d'instructions sur lesquels repose la sémantique du programme. Par exemple, dans le cas considéré dans ce paragraphe, on note que *fnord* utilise deux fois plus les instructions de boucle et manipule deux fois plus de pointeurs que [SSMTP]. La même observation peut être faite à propos du nombre d'invariants générés. À nouveau, *fnord* et *ssmtp*, bien que de taille comparable, produisent chacun une analyse complètement différente. En effet, on obtient un taux d'instrumentation en terme de mécanismes de détection respectivement de 3% et de 75%. Là encore, il faut considérer la sémantique du programme pour comprendre cet écart, notamment le fait qu'un plus grand recours à des multiples indirections via des pointeurs va rendre plus difficile la tâche de l'analyse de valeur.

Cet écart est encore plus grand si on compare le nombre de cibles potentielles (c'est-à-dire l'ensemble des variables injectables avant un appel de fonction) et le nombre de cibles vérifiées (c'est-à-dire l'ensemble des variables contrôlées par un invariant avant un appel de fonction). Le taux de couverture des invariants est le rapport entre ces deux données. Dans le cas de *fnord* et de *ssmtp*, celui-ci est respectivement de 0,7%s et 97,7%. Notons que, si dans cet exemple l'écart de performance en terme d'invariants générés est extrême, il est possible, comme le montre les tableaux 4.1 et 4.2, d'obtenir des taux d'instrumentation intermédiaires.

Notons également grâce au tableau 4.2 que si dans le cas des mécanismes de détection le taux d'instrumentation peut grandement varier, ce n'est pas le cas pour les mécanismes d'injection. En effet, on obtient pour ces derniers un taux d'instrumen-

tation toujours supérieur ou égal à 80%. Ceci montre comme on pouvait s'en douter que la difficulté de calculer des invariants ne réside pas dans le calcul des dépendances mais dans le calcul des domaines de variation (voir les sections 2.3.2 et 2.3.3). Comme la durée d'analyse, le nombre d'invariants générés par notre approche dépend grandement de la sémantique du code source. Il n'est donc pas possible de généraliser les résultats obtenus sur ces exemples.

4.2.1.2 Surcharge à l'exécution

SPEC CPU2006 [SCPU06] est un test de performance pour évaluer les capacités de calcul des processeurs informatiques. Celui-ci repose sur un ensemble de programmes à compiler pour l'architecture concernée ainsi que sur un ensemble de jeux de tests intensifs pour chacun de ces programmes. Ce test de performance est maintenu par une association à but non lucratif nommée *SPEC* (pour *Standard Performance Evaluation Corporation*). Il s'agit d'un test standard dans l'industrie. Son utilisation pour évaluer la surcharge à l'exécution d'un processus d'instrumentation donne la possibilité de comparer objectivement notre approche avec d'autres travaux reposant également sur l'instrumentation. Nous avons sélectionné 6 programmes parmi ceux proposés par le test de performance. Il s'agit des programmes dont le code source est entièrement écrit en *C* (certains programmes peuvent contenir du *C++* ou du *FORTRAN*). Nous avons également éliminé de cette sélection le programme *GCC* afin de ne conserver que des programmes de taille réduite.

	lbm	mcf	libquantum	bzip2	milc	sjeng
Score non-instrumenté	19.24	17.80	23.62	16.59	12.64	18.41
Score instrumenté	19.17	17.72	23.45	16.54	12.59	18.24
Surcharge à l'exécution	0.37%	0.45%	0.72%	0.30%	0.40%	0.93%

TABLE 4.3 – Résultat de l'analyse de la surcharge à l'exécution

Nous avons tout d'abord procédé un test de performance avec 10 itérations pour ces 6 programmes. Nous avons ensuite instrumenté ces programmes à l'aide de l'outil que nous avons développé. Notons que seuls les mécanismes de détection ont été ajoutés aux différents codes source. Puis, nous avons lancé un nouveau test de performance identique au précédent. Dans les deux cas la phase de tests a duré environ 700 minutes. Le tableau 4.3 présente les résultats de ces deux tests de performance. On peut voir que la surcharge à l'exécution induite par nos mécanismes de détection est très faible. Ce résultat était attendu. En effet, on ajoute environ 2% à 3% d'instructions aux programmes et ces instructions ne reposent que sur des opérations booléennes qui sont de faibles consommatrices de cycles processeur. Notons que durant les tests de performance, à aucun moment nous n'avons observé de faux positifs. Il s'agit là également de la confirmation d'un résultat qui était attendu, ceci est en effet garanti par la sur-approximation dont fait preuve le calcul des invariants.

4.2.2 Évaluation du taux de couverture

Nous avons réalisé à l'aide de la plateforme présentée dans ce chapitre deux campagnes de tests. Nous avons choisi de tester deux applications qui remplissent la même tâche, celle de serveur *web*. Il s'agit des serveurs *ihttpd* [IHTTP] et *null-httpd* [NULLD]. Dans cette section, nous allons d'abord expliquer comment nous mesurons la déviation comportementale afin d'évaluer la pertinence des injections réalisées pendant les campagnes de tests. Puis, nous présentons les résultats des deux campagnes de tests avant de proposer une analyse de ces derniers.

4.2.2.1 Détection de la déviation comportementale

Nous avons vu dans la section 4.1.2.2 que nous considérons trois états possibles pour l'arrêt d'un scénario d'exécution : la terminaison correcte, l'interruption inattendue ou le blocage. Dans les deux derniers cas, si les scénarios ont été écrits pour ne jamais provoquer de défaillance par arrêt ni par blocage en situation de fonctionnement normal, on sait automatiquement que le comportement du programme a dévié lorsque la situation se présente. Cette tâche est déjà plus difficile lorsque le scénario d'exécution s'est terminé correctement.

Dans ce cas, une première approche consiste à comparer les résultats produits par le scénario avec des données de référence. Il peut par exemple s'agir d'informations envoyées sur la sortie standard ou enregistrées dans un fichier. Bien sûr, ceci n'est réalisable que si les scénarios de tests ont été écrits de façon à ce que les résultats produits soient identiques ou bien de façon à ce que l'on soit capable de gommer les différences de comportement d'une exécution à l'autre. Cependant, on peut envisager la situation où, bien que le comportement du programme a effectivement dévié, cela n'a pas influencé les données que l'on compare.

C'est pourquoi nous proposons d'aller plus loin et de comparer les traces d'exécution aux traces de référence enregistrées durant la phase préalable à la campagne de tests (voir section 4.1). Cependant, il n'est pas possible cette fois d'écrire les scénarios de façon à ce que les traces d'exécution soient identiques. D'une part, parce que les mécanismes de détection et les mécanismes d'injection que nous avons ajoutés aux programmes vont se comporter différemment entre la phase préalable et la phase de tests. Toutefois, ces variations sont facilement identifiables. En effet, il est facile d'identifier les appels systèmes effectués par notre instrumentation, soit parce qu'ils contiennent un paramètre caractéristique (par exemple une chaîne de caractère commençant par *SIDAN*), soit parce qu'ils apparaissent dans des séquences propres aux mécanismes ajoutés.

D'autre part, même après avoir traité les différences dues à notre instrumentation, les traces d'exécution ne sont pas encore identiques. En effet, deux exécutions d'un même scénario ne vont pas produire exactement les mêmes traces d'exécution et ce même si les deux exécutions ont été réalisées en situation de fonctionnement normal. Par exemple, le nombre d'opérations de lecture d'une *socket*, bien que la longueur

des données à réceptionner soit constante, peut varier d'une exécution à l'autre. Ces différences peuvent apparaître dans l'enchaînement des appels système mais aussi dans leurs arguments.

Pour résoudre ce problème, nous proposons d'utiliser un autre système de détection d'intrusion applicatif. Plus précisément, nous proposons d'utiliser une approche qui contrôle les séquences des appels système en tenant compte de leurs arguments. Ce type d'approche a été présenté dans la section 1.2.2.2. Il s'agit notamment des travaux de Kruegel et al. [KMVV03]. C'est sur la base de ces travaux qu'a été développé le système de détection d'intrusion *Syscall Anomaly* [SYSA]. À l'origine, il s'agit d'un système de détection hors-ligne qui analyse les traces d'exécution fournies par *Snare* [SNARE] pour les systèmes *Linux* et par *BSM* [BSM] pour les systèmes *Solaris* et compatibles.

Pour que les différences présentes dans les traces d'exécution ne soient plus un problème, ce système de détection d'intrusion repose sur deux choses. Tout d'abord, comme toutes les approches qui reposent sur le contrôle des séquences d'appels système, c'est le choix de la taille des séquences qui permet de ne pas lever de fausses alertes malgré la présence de différences dans l'enchaînement des appels système (voir section 1.2.1.2). Ensuite, pour éliminer les différences dues aux arguments des appels système, un profil de comparaison est créé pour chacun d'entre eux (voir section 1.2.2.2).

Nous avons modifié *Syscall Anomaly* [SYSA] pour qu'il fonctionne avec les traces d'exécution fournies par *Strace* [STRAC]. Pour le choix de la taille des séquences lors de la phase d'apprentissage, nous proposons d'utiliser une valeur élevée. En théorie, une valeur élevée augmente le risque de faux positifs. Cependant, nous utilisons cette approche dans un contexte où elle n'a pas à apprendre toutes les traces d'exécution possibles du programme, mais seulement celles qui correspondent à un scénario d'exécution précis.

Dans ce contexte, si nous lui donnons suffisamment de traces d'exécution de référence pour la phase d'apprentissage, l'utilisation d'une valeur élevée pour la taille des séquences des appels système ne provoquera pas de faux positifs. Une alerte nous permettra donc d'identifier à coup sûr une déviation comportementale. L'absence de faux positifs est de toute façon quelque chose dont on s'assure avant la phase d'analyse grâce aux traces d'exécution de référence. En effet, une partie d'entres elles sont utilisées pour la phase d'apprentissage tandis que les autres permettent de tester le système de détection d'intrusion et donc de vérifier qu'aucune alerte n'est levée en situation de fonctionnement normal.

4.2.2.2 Présentation et analyse des résultats

Parmi les logiciels que nous avons instrumentés, nous en avons choisi deux, à savoir *ihttpd* et *nullhttpd*, pour réaliser les campagnes de tests. Ce sont tous les deux des serveurs *web* et leur taux respectif d'instrumentation en terme de mécanismes

d'injection sont proches (voir tableau 4.2). Dans le cas de *ihttpd*, il nous a fallu écrire 33 scénarios pour couvrir 89% des mécanismes d'injection et nous avons réalisé 59840 tests. Dans le cas de *nullhttpd*, il nous a fallu écrire 8 scénarios pour couvrir 81% des mécanismes d'injection et nous avons réalisé 96925 tests. Notons qu'à nouveau, ces campagnes de tests nous ont confirmé que notre approche pour la détection ne génère pas de faux positif. Le tableau 4.4 présente une vue d'ensemble des résultats de ces campagnes.

	ihttpd	nullhttpd
Nombre d'injections réalisées	56584	96925
Nombre d'injections détectées	6419	29107
Taux de détection	11.3%	30.3%
Nombre d'alertes levées	27938	1072310
Nombre moyen d'alertes par détection	4.35	36.84
Nombre de détection par propagation	2	174
Taux de propagation	0.03%	0.59%
Distance moyenne de propagation	87	1.80

TABLE 4.4 – Résultat des campagnes de tests

Considérons tout d'abord la première mesure du taux de faux négatifs que nous souhaitons réaliser, à savoir l'ensemble des injections qui n'ont pas été détectées par notre approche. Dans le cas de *ihttpd*, nous obtenons un taux de détection de 11,3%. Dans le cas de *nullhttpd*, nous obtenons un taux de détection de 30,3%. Ceci correspond à un taux de faux négatifs respectivement de 88,66% et 69,97%. Ces résultats sont à mettre en relation avec la couverture des invariants de ces deux programmes (respectivement 29,1% et 56,1%, voir tableau 4.2). Les taux de détection sont cohérents avec les taux de couverture en terme d'invariants : le programme qui vérifie la valeur d'un plus grand nombre de cibles potentielles est également celui qui a le meilleur taux de détection.

Cependant, cet écart dans le nombre de cibles potentielles vérifiées n'explique pas certaines différences. Par exemple, le nombre d'alertes levées en moyenne lorsqu'une injection est détectée. Certes, à nouveau le programme qui vérifie le plus d'invariants est celui qui lève le plus d'alertes. Toutefois, si *nullhttpd* vérifie environ deux fois plus d'invariants, il lève aussi par contre en moyenne presque 8,5 fois plus d'alertes. Nous avons dit dans la section 4.2.1.1 que les résultats de l'instrumentation dépendent de la sémantique du programme et que ces derniers seront différents pour chaque programme. Ceci est également vrai pour les résultats de la détection : en situation d'attaque, le comportement d'un programme durci avec nos mécanismes de détection dépend de la sémantique du programme original.

On observe également cela avec la propagation de l'erreur : il y a une grande variation aussi bien en terme de distance que de taux de propagation. Certes, dans les deux cas, on peut voir que la propagation est très faible. 0.03% pour *ihttpd* et 0.6% pour *nullhttpd*. Ces chiffres nous permettent de supposer que, dans le cas général, la propagation des erreurs au sein des programmes n'apporte qu'une aide très limitée à notre approche pour la détection. La couverture de détection de l'approche que nous proposons repose donc essentiellement sur la capacité de l'analyse statique à calculer des invariants.

Comparons maintenant les résultats de notre approche avec le système de détection d'intrusion analysant les données fournies par *strace* et que nous utilisons pour détecter les éventuelles déviations comportementales au niveau des appels système (voir section 4.2.2.1). Ajoutons également à la comparaison la simple détection de l'injection lorsque celle-ci a engendré une défaillance (par arrêt ou par blocage). Le tableau 4.5 présente ces informations pour *ihttpd* et *nullhttpd*. On peut voir que dans tous les cas notre approche pour la détection est celle qui détecte le plus souvent une injection (respectivement 11,34% et 30,03%). L'écart n'est toutefois pas très important. Cependant, rappelons que ce dernier nécessite une phase d'apprentissage préalable à la détection et que nous l'avons placé dans une situation où celle-ci maximise ses capacités de détection.

		coninva	*syscall-anomaly*	défaillance
ihttpd	Injection détectées	6419	4383	1167
ihttpd	Taux de détection	11,34%	7,74%	2,06%
nullhtpd	Injection détectées	29107	21337	9184
nullhtpd	Taux de détection	30,03%	22,01%	9,47%

TABLE 4.5 – Comparaison des résultats de la détection

Cependant, en pratique, les utilisateurs malveillants cherchent en priorité à violer les propriétés de confidentialité et d'intégrité. Mettre le système ciblé dans un état de défaillance par arrêt ou par blocage n'est bien souvent l'objectif que si la vulnérabilité exploitée ne permet que cela. Nous allons donc maintenant considérer les résultats de la détection pour *coninva* et pour *syscall-anomaly* en distinguant le cas où durant les campagnes de tests les scénarios ont été exécutés jusqu'à leur terme et le cas où ces derniers se sont soldés par une défaillance par arrêt ou par blocage du serveur. Ces informations sont présentées dans le tableau 4.6 (la défaillance implique uniquement le blocage ou l'arrêt).

Dans le cas de *ihttpd*, on note peu de différence dans les taux de détection entre le cas général et le cas où le programme est dans un état non-défaillant. Ceci est cohérent avec le fait que les cas de défaillance par arrêt ou par blocage ne représentent qu'une faible proportion des tests (2,06%, voir tableau 4.5). Dans le cas de *nullhttpd*, les cas de défaillance par arrêt ou par blocage représentent une plus grande propor-

tion (9,47%, voir tableau 4.5). De plus, cette fois la déviation comportementale est plus souvent détectée quand le programme est dans un état d'arrêt ou de blocage lors des tests. L'écart se creuse donc entre *coninva* et *syscall-anomaly* dans le cas de *nullhttpd*. À nouveau on note donc que les effets d'une campagne d'injection varient d'un programme à l'autre.

Nous avons donné le taux de faux négatifs de notre approche pour la détection dans le cas de l'ensemble des injections réalisées. Il est respectivement de 88,66% et 69,97% pour *ihttpd* et *nullhttpd*. Ce taux est un premier résultat important pour l'évaluation de notre système de détection d'intrusion. En effet, si un état erroné du programme n'a pas provoqué déviation comportementale observable par *syscall-anomaly*, il ne nous ait pas pour autant possible d'en conclure que cet état erroné ne permet pas de réaliser une intrusion. Prenons comme exemple l'attaque contre la variable de boucle dans le code inspiré de la version vulnérable de *openssh* (voir la section 2.3.1.2). Une injection qui reproduirait l'état erroné du programme nécessaire à la réalisation de cette attaque ne provoquerait pas de défaillance par arrêt ou par blocage ni ne serait détectée par *syscall-anomaly*. C'est pourquoi nous considérons que les estimations obtenues précédemment sont des sur-approximations du taux de faux négatifs réels.

Cependant, dans le cas de *syscall-anomaly*, la déviation comportementale étant détectée à partir d'un modèle de référence construit spécifiquement pour le scénario utilisé pour réaliser l'injection (voir section 4.2.2.1), on peut raisonnablement penser que le taux de faux positifs est nul dans cette situation. Ceci est d'ailleurs vérifié dans la phase préalable à la phase de tests (voir section 4.2.2.1). Lorsqu'une alerte est effectivement levée par *syscall-anomaly*, nous choisissons de considérer en conséquence que l'état erroné du programme obtenu par injection permet effectivement de réaliser une intrusion. Si dans ce cas notre approche n'a pas levé d'alerte, nous considérons donc également qu'il s'agit d'un faux négatif grave. L'ensemble de ces

(a) *ihttpd* et *coninva*

	Fonctionnel	Défaillant
Détection	6404	15
Pourcentage	11,55%	1,28%

(b) *ihttpd* et *syscall-anomaly*

	Fonctionnel	Défaillant
Détection	4370	13
Pourcentage	7,88%	1,11%

(c) *nullhttpd* et *coninva*

	Fonctionnel	Défaillant
Détection	26945	2162
Pourcentage	30,70%	23,54%

(d) *nullhttpd* et *syscall-anomaly*

	Fonctionnel	Défaillant
Détection	16970	4367
Pourcentage	19,34%	47,55%

TABLE 4.6 – Comparaison des résultats de la détection suivant l'état de fin du scénario d'exécution

	ihttpd	*nullhttpd*
Nombre d'injections réalisées	56584	96925
Alertes *coninva*	5655	9705
Taux de détection	10,00%	10,01%
Alertes *syscall-anomaly*	3621	5539
Taux de détection	6,40%	5,71%
Alertes *coninva* et *syscall-anomaly*	749	17240
Taux de détection	1,32%	17,79%

TABLE 4.7 – Résultats de la détection pour une terminaison correcte des scénarios d'exécution

absences d'alerte malgré la présence d'une déviation comportementale nous permet de donner une nouvelle estimation du taux de faux négatifs de notre approche. Il ne s'agit par contre pas d'une sur-approximation cette fois.

Les informations qui sont nécessaires à cette estimation sont présentées dans le tableau 4.7. Les faux négatifs malgré la mise en évidence d'une déviation comportementale correspondent au cas où seul *syscall-anomaly* a levé une alerte. Dans le cas de *ihttpd*, avec 3621 injections non détectées par notre système de détection d'intrusion, cela correspond à un taux de faux négatifs de 6,40%. Dans le cas de *nullhttpd*, avec 5539 injections non détectées cette fois, cela correspond à un taux de faux négatifs de 5,71%.

4.3 Résumé et discussion

Dans ce chapitre, nous avons tout d'abord présenté une plateforme de tests qui permet d'automatiser l'évaluation d'un mécanisme de détection d'intrusion donné pour un programme particulier. Cette plateforme repose sur le mécanisme d'injection de fautes que nous avons présenté au chapitre 3 mais aussi sur un moniteur d'exécution que nous avons développé. Ce moniteur gère les multiples exécutions du programme nécessaires à la campagne de tests. C'est également ce moniteur qui contrôle les mécanismes d'injection et qui se charge de la collecte des informations nécessaires à l'estimation du taux de faux négatifs du mécanisme de détection d'intrusion en cours de test.

Afin d'utiliser cette plateforme de tests pour un programme donné, un ensemble de scénarios d'exécution doit être fourni au moniteur d'exécution. La difficulté consiste ici à fournir un jeu de scénarios ayant une bonne couverture de code pour le programme considéré. Nous avons choisi de mesurer cette couverture en fonction des points d'injection atteignables dans le programme. Nous avons présenté la difficulté d'écrire manuellement des scénarios qui permettent d'obtenir un bon taux de couverture et nous avons présenté les cas où il est difficile voir impossible de couvrir

des points d'injection particuliers.

Puis, nous avons présenté dans ce chapitre l'évaluation de notre approche pour la détection. Nous avons d'abord donné les performances de l'instrumentation. L'outil que nous avons implémenté pour tester notre modèle de détection sur les programmes écrits en langage *C* a pu être appliqué avec succès sur plusieurs cas réels. Ces applications montrent que la phase d'analyse du code source peut être longue et que le résultat en terme d'invariants générés peut grandement varier suivant la sémantique des programmes analysés. En utilisant un test de performance, nous avons confirmé un résultat qui était attendu : notre approche pour la détection ne génère aucun faux positif et la surcharge à l'exécution est vraiment très faible.

Nous avons ensuite présenté les résultats des campagnes de tests réalisées grâce à la plateforme que nous avons développée. Nous avons tout d'abord pu constater qu'à l'instar de la performance de l'instrumentation, la performance de la détection dépend elle aussi de la sémantique du programme original. Néanmoins, ces résultats nous ont permis de donner une estimation du taux de faux négatifs de notre approche pour la détection que nous considérons être une sur-approximation du taux réel. Nous avons également donné une estimation du taux de faux négatifs pour les simulations d'attaque que nous considérons comme étant les plus susceptibles d'être représentatives des attaques réels. Dans les deux cas, ceci nous a permis de montrer que notre système de détection d'intrusion est une approche valable pour détecter les attaques contre les données de calcul. Notons qu'à nouveau les campagnes de tests n'ont présenté aucun faux positifs. Toutefois, ces résultats nous ont aussi permis de constater un second point important : la propagation de l'erreur n'apporte que peu d'aide à notre mécanisme de détection qui, au final, repose en grande partie sur la vérification des invariants portant directement sur les variables ciblées par l'utilisateur malveillant.

Conclusion

Ces travaux de thèse ont pour objectif la détection des intrusions issues des attaques contre les données de calcul. Cette catégorie de données désigne les informations contenues dans les emplacements mémoire utilisés par les processus pour stocker la valeur des variables exprimées dans leur code source originel respectif. Nous proposons dans ces travaux de recherche un modèle de détection d'intrusion ciblant spécifiquement les attaques contre les données de calcul. Son objectif est de vérifier des invariants portant sur les données de calcul qui constituent l'ensemble des cibles potentielles pour un utilisateur malveillant. Comme il s'agit d'une approche de type comportementale, ceci donne à notre modèle la capacité de détecter des intrusions encore inconnues au moment de sa construction (ce que ne permet pas les approches par signature).

Nous avons proposé une implémentation de ce modèle de détection dans le cadre des programmes écrits en langage *C*. Celle-ci repose sur un analyseur statique de code, *Frama-C*, et est donc utilisable sur des programmes réels. Les invariants obtenus portent directement sur les variables du code source. Ces derniers sont ensuite traduits sous forme d'assertions exécutables avant d'être ajoutées au code source des applications. Notre système de détection d'intrusion est donc une approche en boîte blanche de niveau applicatif qui repose sur un modèle orienté autour des variables. De plus, les techniques d'analyse statique utilisées par *Frama-C* nous assurent que le modèle de détection obtenu est une sur-approximation du comportement réel du programme. Ceci nous permet d'affirmer que notre système de détection d'intrusion est complet, c'est-à-dire qu'il ne génère pas de faux positifs. Grâce à cette implémentation, nous avons pu tester la validité de notre approche pour la détection.

Nous avons pu ainsi vérifier que notre système de détection d'intrusion détecte bien des cas réels d'attaques bien que la construction du modèle de détection ne repose pas sur leur connaissance. Nous avons également pu vérifier que celui-ci ne génère bien aucun faux positif. De plus, nous avons pu voir que notre système de

détection d'intrusion n'induit que très peu de surcharge à l'exécution. Ceci fait que notre approche pour la détection ne peut être qu'un atout pour le système sur lequel elle est déployée. Toutefois, l'inconvénient de notre méthode est que seuls les comportements invariants des applications sont saisis par le modèle. Par conséquent, certains comportements des applications ne peuvent être modélisés, et la méthode peut présenter des faux négatifs. Ceci est d'autant plus vrai que nous ne pouvons aucunement garantir que tous les comportements invariants des applications ont bien été saisis.

Pour tester cela, nous avons aussi proposé un modèle de simulation des attaques contre les données de calcul. Cette approche combine un modèle de faute qui reproduit les caractéristiques de ce type d'attaque avec des mécanismes d'injection embarqués dans les applications par un processus d'instrumentation similaire à celui utilisé par les mécanismes de détection. Notre approche pour l'évaluation présente l'avantage de générer de manière automatique des états internes erronés pour les applications testées. Ceci peut alors être utilisé pour obtenir une estimation du taux de faux négatifs d'un système de détection d'intrusion vis-à-vis des attaques contre les données de calcul. Nous avons donc également développé une plateforme de tests autour de ces mécanismes d'injection. Bien que la procédure de tests et l'analyse des résultats soient entièrement automatisées, les scénarios d'exécution nécessaires pour mener les campagnes d'injection restent à écrire manuellement. Toutefois, nous avons pu ainsi vérifier que les capacités de détection de notre approche pour la détection dépendaient bien du nombre d'invariants contrôlés par les programmes. De plus, nous avons pu aussi voir que la propagation de l'erreur au sein des processus n'apportait que peu d'aide à notre système de détection d'intrusion.

Nous proposons ici un résumé de l'ensemble de nos travaux avant de terminer par des perspectives pour envisager d'étendre nos contributions.

Résumé

Le chapitre 2 présente la première partie importante de nos travaux : l'approche pour la détection que nous proposons. Nous avons tout d'abord expliqué les caractéristiques des attaques contre les données de calcul et en quoi ces dernières se distinguent des autres types d'attaque. Ceci nous a notamment permis de montrer que pour perpétuer une intrusion, un utilisateur malveillant va chercher à cibler un ensemble bien précis de données de calcul. À l'aide de la logique de Hoare, nous avons ensuite expliqué que le code source des applications peut contenir des informations qui peuvent être utilisées pour détecter ce type bien précis d'attaque. Nous avons détaillé cela sur un exemple d'exploitation de vulnérabilité.

Puis, nous avons présenté notre modèle de détection. Nous l'avons tout d'abord présenté empiriquement sur un cas réel d'attaques contre les données de calcul. Pour

cela, nous avons détaillé la vulnérabilité utilisée dans notre exemple ainsi que les différents scénarios d'attaque et comment des invariants portant sur certaines variables permettent de détecter ces attaques. Enfin, nous avons présenté formellement notre modèle de détection. Celui-ci correspond à l'ensemble des domaines de variation des variables qui influencent l'exécution des appels de fonction. Ces domaines de variation sont calculés juste avant les appels de fonction et uniquement pour les variables qui sont atteignables à ces endroits du code source.

Nous avons ensuite présenté une méthode pour construire un tel modèle. Premièrement, nous proposons d'utiliser le graphe de dépendance du programme pour déterminer pour chaque appel de fonction l'ensemble des variables qui influencent son exécution. Deuxièmement, nous proposons d'utiliser l'interprétation abstraite pour calculer pour chacun de ces ensembles de variables leur domaine de variation. Pour finir, nous présentons une implémentation de notre approche que nous avons réalisée pour les programmes écrits en langage *C*. Nous détaillons d'abord la phase de construction du modèle qui repose sur un outil d'analyse statique existant, *Frama-C*. Nous détaillons ensuite la phase d'instrumentation, celle-ci ayant pour contrainte de ne pas modifier le processus original de compilation.

Le chapitre 3 présente la seconde partie importante de nos travaux : l'approche pour l'évaluation que nous proposons. Nous commençons par aborder la problématique de la simulation des erreurs engendrées par les attaques contre les données de calcul. Pour cela, nous présentons d'abord le modèle de faute que nous proposons pour simuler ce type bien particulier d'attaques. Nous étudions les caractéristiques qui doivent être simulées, quel sera leur impact sur le programme et dans quel cas ces dernières peuvent être détectées. Nous expliquons ensuite comment nous proposons de construire notre modèle de simulation. La principale problématique ici est de savoir comment déterminer l'ensemble des cibles potentielles. Il s'agit du même ensemble de variables que pour la détection. Nous proposons donc à nouveau de nous reposer sur le graphe de dépendance du programme et d'embarquer les mécanismes d'injection au sein des applications.

Nous expliquons ensuite comment notre modèle de faute peut être utilisé pour l'évaluation d'un système de détection d'intrusion. Nous posons comme objectif que le résultat obtenu doit être une sur-approximation du taux de faux négatifs réel. Cela implique que nous voulons placer le système de détection d'intrusion à évaluer dans la situation la moins favorable possible. Pour respecter cette contrainte, nous montrons que notre modèle de faute doit être utilisé pour simuler une intrusion qui ne nécessite qu'une seule exploitation de la vulnérabilité, que la vulnérabilité donne accès à l'ensemble de l'espace mémoire du processus et que l'exploitation ne vise qu'une seule variable. Nous présentons enfin les modifications que nous avons apportées à notre outil afin qu'il instrumente aussi les programmes pour l'injection et comment les mécanismes d'injection ainsi ajoutés doivent être utilisés.

Le chapitre 4 présente la dernière partie de nos travaux : l'évaluation de notre système de détection d'intrusion, notamment à l'aide de notre modèle de simulation

d'attaque. Nous commençons par présenter la plateforme de tests que nous avons développée autour de nos mécanismes d'injection. Il s'agit d'une plateforme qui automatise la réalisation de tests ainsi que l'analyse des résultats obtenus. Nous abordons tout d'abord les problématiques d'écriture des scénarios d'exécution et de collecte des informations. Les scénarios doivent permettre de couvrir suffisamment le code des programmes utilisés pour les tests. Nous avons choisi de mesurer ce taux de couverture en fonction des appels de fonction. Les informations collectées sont utilisées pour produire deux résultats : une sur-approximation du taux réel de faux négatifs et une évaluation du taux de détection pour les injections ayant provoqué une déviation comportementale.

Pour finir, nous présentons les résultats de l'évaluation de notre système de détection d'intrusion. Nous commençons par donner les performances de l'analyse. On note que la durée d'analyse peut être très grande, notamment en fonction de la taille du code à analyser, mais qu'en fonction de la sémantique du code, deux programmes de taille similaire peuvent présenter des durées d'analyse complètement différentes. Puis, nous donnons le niveau de surcharge à l'exécution. On note que la surcharge induite par nos mécanismes de détection est très faible, toujours inférieure à 1%. Nous continuons avec les performances de la détection. Nous pouvons voir que les résultats de la détection varient grandement d'un programme à l'autre, malgré un taux d'instrumentation similaire. Ce qui change, c'est le nombre d'invariants vérifiés. On voit ici la limite de notre approche : si la sémantique du code original ne permet pas de calculer suffisamment d'invariants, l'efficacité de notre approche sera alors limitée. De plus, la propagation de l'erreur n'apporte que peu d'aide à notre modèle de détection. Dans tous les cas, nous avons pu vérifier que notre approche ne génère bien pas de faux positif.

Perspectives

Les résultats de ces travaux ouvrent des perspectives qu'ils conviendraient d'étudier en complément, notamment dans les directions suivantes :

– Il serait intéressant d'explorer d'autres méthodes pour la génération des invariants. En effet, nous avons vu que la propagation de l'erreur est relativement limitée et que les capacités de détection de notre approche reposent principalement sur la vérification des variables directement ciblées par les attaques. On peut par exemple envisager que d'autres méthodes d'analyse statique puissent nous fournir des invariants sur d'autres types de données que les entiers (par exemple, sur les chaînes de caractères) ou nous fournir d'autre type d'invariants sur les entiers (par exemple, une variable doit rester constante sur l'ensemble d'un bloc de code ou encore être égale à la somme de deux autres variables, quand bien même le domaine de variation de toutes ces variables n'est pas statiquement calculable [RCK04, RCK07]). Cependant, les durées d'analyse étant

déjà très grandes actuellement, il conviendra d'étudier la faisabilité d'utiliser ces méthodes supplémentaires dans le cadre de programmes réels.

- On pourrait aussi envisager d'utiliser d'autres méthodes d'analyse statique qui seraient moins performantes en terme d'invariants générés mais qui demanderaient beaucoup moins de ressources pour conduire leur analyse [BDBD96, BBM97]. En effet, bien que les programmes auxquels nous avons appliqué notre méthode pour la détection soient bien des cas réels, leur code source respectif sont toutefois de taille modeste (inférieure à 15 000 lignes de code). Pourtant, l'analyse peut déjà être parfois très longue (jusqu'à plusieurs jours). De telles méthodes pourraient nous permettre d'envisager d'appliquer notre approche à des programmes de taille bien supérieure. Il conviendra cette fois d'étudier l'impact sur la détection de ces analyses moins précises.
- Nous avons vu que le masquage de faute peut limiter les capacités de détection de notre approche, notamment lorsque les variables qui influencent les appels de fonction sont réinitialisées par le programme entre le moment où la variable est utilisée et le moment ou l'appel de fonction est exécuté. Il serait intéressant de voir si un prétraitement du code source qui nous assure que chaque variable n'est affectée par le programme qu'une seule fois (*Static Single Assignement* ou *SSA*) [CFR+91, BCHS98] pourrait améliorer la précision de nos invariants.
- Bien que l'instrumentation nécessaire à notre approche pour l'évaluation soit complètement automatisée, pour réaliser des campagnes de tests, les scénarios d'exécution doivent être écrits manuellement. Il serait intéressant d'explorer la possibilité de coupler notre approche pour la simulation d'attaque avec des techniques utilisées en test logiciel. On peut par exemple envisager d'avoir recours à des techniques similaires au *fuzzing* [SGA07] pour nous assister dans l'écriture des scénarios d'exécution.

Bibliographie

[AAC+03] André Adelsbach, Dominique Alessandri, Christian Cachin, Sadie Creese, Yves Deswarte, Klause Kursawe, Jean-Claude Laprie, David Powell, Brian Randell, James Riordan, Peter Ryanand William Simmonds, Robert Stroud, Paulo Veríssimo, Michael Waidner, and Andreas Wespi. Conceptual model and architecture of MAFTIA. MAFTIA deliverable d21, LAAS-CNRS and University of Newcastle upon Tyne, January 2003.

[ABC+95] Jean Arlat, Jean-Paul Blanquart, Alain Costes, Yves Crouzet, Yves Deswarte, Jean-Charles Fabre, Hubert Guillermain, Mohamed Kaâniche, Karama Kanoun, Jean-Claude Laprie, Corinne Mazet, David Powell, and Christophe Rabéjac et Pascale Thévenod. *Guide de la Sûreté de Fonctionnement*. Cépaduès - Éditions, 1995.

[ABEL05] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *CCS '05 : Proceedings of the 12th ACM conference on Computer and communications security*, 2005.

[ADAFE] Autodafé, Martin Vuagnoux.
http ://autodafe.sourceforge.net/.

[And80] James P. Anderson. Computer Security Threat Monitoring and Surveillance. Technical report, James P. Anderson Company, Fort Washington, Pennsylvania, April 1980.

[ASF] Apache HTTP Server, Apache Software Foundation.
http ://www.apache.org/.

[BBM97] Nikolaj Bjørner, Anca Browne, and Zohar Manna. Automatic generation of invariants and intermediate assertions. *Theor. Comput. Sci.*, 173 :49–87, February 1997.

[BCHS98] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction

of static single assignment form. *Softw. Pract. Exper.*, 28 :859–881, July 1998.

[BCNP03] Alfredo Benso, Stefano Di Carlo, Giorgio Di Natale, and Paolo Prinetto. A watchdog processor to detect data and control flow errors. *IEEE International On-Line Testing Symposium*, 0 :144–148, 2003.

[BCS06] Sandeep Bhatkar, Abhishek Chaturvedi, and R. Sekar. Dataflow anomaly detection. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, 2006.

[BDBD96] Matt Bishop, Michael Dilger, Matt Bishop, and Michael Dilger. Abstract checking for race conditions in file accesses. *Computing Systems*, 9(2) :131–152, Spring 1996.

[BK85] William Earl Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings of the 14th Annual Computer Security Applications Conference (ACSAC'98)*, pages 18–27, December 1985.

[BL73] D. Elliott Bell and Leonard J. LaPadula. Secure computer systems : Mathematical foundations. MTR-2547 (ESD-TR-73-278-I) Vol. 1, MITRE Corp., Bedford, 1973.

[BL76] D. Elliott Bell and Leonard J. LaPadula. Secure computer system : Unified exposition and multics interpretation. Mtr-2997 (esd-tr-75-306), MITRE Corp., 1976.

[BM82] Robert S. Boyer and J. Strother Moore. A mechanical proof of the unsolvability of the halting problem. *Journal of ACM*, 31 :441–458, 1982.

[BNS+06] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards automatic generation of vulnerability-based signatures. In *SP '06 : Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 2–16, Washington, DC, USA, 2006. IEEE Computer Society.

[BR01] B. Balajinath and S.V. Raghavan. Intrusion detection through learning behavior model. *Computer Communications*, 24(12) :1202–1212, July 2001.

[BSM] Basic Security Module, TrustedBSD.
http ://www.trustedbsd.org/openbsm.html.

[Bul07] Yuriy Bulygin. Remote and Local Exploitation of Network Drivers. Black Hat USA, 2007.

[But07] Laurent Butti. Wi-Fi Advanced Fuzzing Wi-Fi Advanced Fuzzing. Black Hat Europe, 2007.

[C0133] Advisory CA-2001-33 Multiple Vulnerabilities in WU-FTPD, CERT.
http ://www.cert.org/advisories/CA-2001-33.html.

[C99] ISO/IEC 9899 :TC3 WG14/N1256, The international standardization working group for the programming language C.
http ://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf.

[Cap75] Michel Caplain. Finding invariant assertions for proving programs. In *Proceedings of the international conference on Reliable software*, pages 165–171, 1975.

[CC76] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*. Dunod, Paris, France, April 1976.

[CC77a] Patrick Cousot and Radhia Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.

[CC77b] Patrick Cousot and Radhia Cousot. Automatic synthesis of optimal invariant assertions : mathematical foundations. In *ACM Symposium on Artificial Intelligence & Programming Languages*, pages 1–12, August 1977.

[CCERT] CompCert, INIRIA.
http ://compcert.inria.fr/.

[CCH06] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13 :451–490, October 1991.

[CO00] Frédéric Cuppens and Rodolphe Ortalo. LAMBDA : A Language to Model a Database for Detection of Attacks. In H. Debar, L. Mé, and S. F. Wu, editors, *Proceedings of the Third International Workshop on the Recent Advances in Intrusion Detection (RAID'2000)*, number 1907 in LNCS, pages 197–216, October 2000.

[COMR] COMRaider, David Zimmer (iDefense Labs).
http ://labs.idefense.com/software/fuzzing.php.

[Cou02] Patrick Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. 277 :47–103, April 2002.

[CP03] Sung-Bae Cho and Hyuk-Jang Park. Efficient anomaly detection by modeling privilege flows using hidden markov model. *Computers and Security*, 22(1) :45–55, January 2003.

[Cri00] Paul J. Criscuolo. Distributed denial of service. Technical Report CIAC-2319, CIAC, Lawrence Livermore National Laboratory, US Department of Energy, February 2000.

[CXS+05] Shuo Chen, Jun Xu, Emre Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *Usenix Security Symposium*, pages 177—-192, 2005.

[DBS92] Hervé Debar, Monique Becker, and Didier Siboni. A neural network component for an intrusion detection system. In *Proceedings of the IEEE Symposium of Research in Computer Security and Privacy*, pages 240–250, Oakland, CA, May 1992.

[DELTA] Delta, Daniel S. Wilkerson, Scott McPeak and Simon Goldsmith. http ://delta.tigris.org/.

[Der99] Renaud Deraison. *The Nessus Attack Scripting Language Reference Guide*, September 1999.

[Dij59] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1 :269–271, 1959.

[DM98] Wenliang Du and Aditya P. Mathur. Vulnerability testing of software system using fault injection. Technical report, Purdue University, April 1998.

[DM00] Wenliang Du and Aditya P. Mathur. Testing for software vulnerability using environment perturbation. In *Proceedings of the 2000 International Conference on Dependable Systems and Networks (DSN 2000), Workshop On Dependability Versus Malicious Faults*, pages 603–612. IEEE Computer Society, June 2000.

[DMTT11] Jonathan-Christofer Demay, Frédéric Majorczyk, Éric Totel, and Frédéric Tronel. Detecting illegal system calls using a data-oriented detection model. In *Proceedings of the 26th International Conference on Information Security (IFIP/SEC 2001)*, 2011.

[DROPB] Dropbear SSH, Matt Johnston. http ://matt.ucc.asn.au/dropbear/dropbear.html.

[DTT09a] Jonathan-Christofer Demay, Éric Totel, and Frédéric Tronel. Automatic software instrumentation for the detection of non-control-data attacks. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection (RAID 2009)*, Saint-Malo, September 2009.

[DTT09b] Jonathan-Christofer Demay, Eric Totel, and Frederic Tronel. Sidan : a tool dedicated to software instrumentation for detecting attacks on non-control-data. In *4th International Conference on Risks and Security of Internet and Systems (CRISIS'2009)*, Toulouse, October 2009.

[DTT10] Jonathan-Christofer Demay, Éric Totel, and Frédéric Tronel. Génération et évaluation de mécanismes de détection d'intrusion au niveau applicatif. In *Actes de la cinquième conférence sur la Sécurité des Architectures Réseaux et Systèmes d'Information SARSSI 2010*, Menton, May 2010.

[Edd07] Wesley M. Eddy. TCP SYN Flooding Attacks and Common Mitigations. RFC 4987, August 2007.

[EFS] Evolutionary Fuzzing System, Jared DeMott (VDA Labs). http ://www.vdalabs.com/tools/efs_gpf.html.

[EVK00] Steven T. Eckmann, Giovanni Vigna, and Richard A. Kemmerer. Statl : An attack language for state-based intrusion detection. In *Proceedings of the ACM Workshop on Intrusion Detection*, November 2000.

[EVK02] Steven T. Eckmann, Giovanni Vigna, and Richard A. Kemmerer. STATL : an attack language for state-based intrusion detection. *Journal of Computer Security*, 10(1-2) :71–103, 2002.

[FFUZZ] FileFuzz, Michael Sutton (iDefense Labs). http ://labs.idefense.com/software/fuzzing.php.

[FHSL96] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A Sense of Self for Unix Processes. In *Proceedings of the 1996 IEEE Symposium on Research in Security and Privacy*, pages 120–128. IEEE Computer Society, IEEE Computer Society Press, May 1996.

[FNORD] fnord, Felix von Leitner. http ://www.fefe.de/fnord/.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9 :319–349, 1987.

[FPLUG] Frama-C, Frama-C Plug-In development guide. http ://frama-c.com/download/plugin-developer-Boron-20100401.pdf.

[FRAMA] Frama-C, Framework for Modular Analysis of C. http ://www.openssh.com/.

[Gli84] Virgil D. Gligor. A note on denial-of-service in operating systems. *IEEE Trans. Softw. Eng.*, 10(3) :320–324, 1984.

[GO08] Jean Goubault-Larrecq and Julien Olivain. A smell of orchids. In Martin Leucker, editor, *Proceedings of the 8th Workshop on Runtime Verification (RV'08)*, volume 5289 of *Lecture Notes in Computer Science*, pages 1–20, Budapest, Hungary, mar 2008. Springer.

[GPF] General Purpose Fuzzing, VDA Labs and Applied Security. http ://www.vdalabs.com/tools/efs_gpf.html, http ://www.appliedsec.com/.

[GRRV03] O. Goloubeva, Maurizio Rebaudengo, Matteo Sonza Reorda, and Massimo Violante. Soft-error detection using control flow assertions. In *Proceedings of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'03)*, 2003.

[GRS04] Debin Gao, Michael K. Reiter, and Dawn Song. Gray-box extraction of execution graphs for anomaly detection. In *Proceedings of the 11th ACM conference on Computer and communications security*, 2004.

[GRS06] Debin Gao, Michael K. Reiter, and Dawn Song. Behavioral distance measurement using hidden markov models. In *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID 2006)*, pages 19–40, Hamburg, Germany, September 2006.

[GSHC09] Kent Griffin, Scott Schneider, Xib Hu, and Tzi Chiueh. Automatic generation of string signatures for malware detection. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection (RAID 2009)*, 2009.

[GSS99a] Anup K. Ghosh, Aaron Schwartzbard, and Michael Schatz. Learning program behavior profiles for intrusion detection. In *Proceedings of the 1st USENIX Workshop on Intrusion Detection and Network Monitoring*, April 1999.

[GSS99b] Anup K. Ghosh, Aaron Schwartzbard, and Michael Schatz. Using program behavior profiles for intrusion detection. In *Proceedings of the 2nd SANS Workshop On Intrusion Detection and Response (ID'99)*, February 1999.

[GSV05] Rajeev Gopalakrishna, Eugene H. Spafford, and Jan Vitek. Efficient intrusion detection using automaton inlining. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2005.

[GTM09] Laurent Georges, Valérie Viet Triem Tong, and Ludovic Mé. Blare tools : A policy-based intrusion detection system automatically set by the security policy. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection (RAID 2009)*, 2009.

[HCF05] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for java. In *Annual Computer Security Applications Conference (ACSAC)*, 2005.

[HFS98] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3) :151–180, 1998.

[Hoa69] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :576–585, October 1969.

[HOM06] William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. Using positive tainting and syntax-aware evaluation to counter sql injection attacks. In *SIGSOFT '06/FSE-14 : Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 175–185, New York, NY, USA, 2006. ACM.

[IHTTP] ihttpd, Ivan Skytte Jørgensen.
http ://i1.dk/download/ihttpd/.

[JL01] Anita Jones and Song Li. Temporal signatures for intrusion detection. In *Proceedings of the 17th Annual Computer Security Applications Conference*, page 252. IEEE Computer Society, 2001.

[JM05] Ranjit Jhala and Rupak Majumdar. Path slicing. In *Proceedings of the 27th Annual ACM Conference on Programming Language Design and Implementation*, pages 38–47, 2005.

[KKA95] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. FERRARI : a flexible-based fault and error injection system. In *IEEE Transactions on Computers*, volume 44, pages 248–260. IEEE Computer Society, February 1995. Special issue on fault-tolerant computing.

[KKMR05] Christopher Kruegel, Engin Kirda, Darren Mutz, and William Robertson. Automating mimicry attacks using static binary analysis. In *Proceedings of the 14th conference on USENIX Security Symposium*, 2005.

[KKP+81] David J. Kuck, Robert Henry Kuhn, David A. Padua, Bruce Leasure, and Michael Joseph Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the Eighth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 207–218, 1981.

[KL88] Bogdan Korel and Janusz Laski. Dynamic program slicing. In *Information Processing Letters*, volume 29, pages 155–163, October 1988.

[KMRV03] Christopher Kruegel, Darren Mutz, William Robertson, and Fredrik Valeur. Bayesian event classification for intrusion detection. In *Proceedings of the 19th Annual Computer Security Applications Conference*. IEEE Computer Society, December 2003.

[KMVV03] Christopher Kruegel, Darren Mutz, Fredrik Valeur, and Giovanni Vigna. On the detection of anomalous system call arguments. In *8th European Symposium on Research in Computer Security (ESORICS 2003)*, pages 326–343, Gjøvik, Norway, October 2003.

[KRL97] Calvin Ko, Manfred Ruschitzka, and Karl N. Levitt. Execution monitoring of security-critical programs in a distributed system : A specification-based approach. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 175–187, Oakland, CA, May 1997.

[LB97a] Terran Lane and Carla E. Brodley. An application of machine learning to anomaly detection. In *Proc. of the 20th National Information Systems Security Conference*, pages 366–380, October 1997.

[LB97b] Terran Lane and Carla E. Brodley. Detecting the abnormal : Machine learning in computer security. Technical Report ECE-97-1, Department of Electrical and Computer Engineering, Purdue University, January 1997.

[LB97c] Terran Lane and Carla E. Brodley. Sequence matching and learning in anomaly detection for computer security. In *Proceedings of the AAAI-97 Workshop on AI Approaches to Fraud Detection and Risk Management*, pages 43–49, July 1997.

[LHF+96] R. Lippmann, J. W. Haines, D. Fried, J. Korba, and K. Das. The 1999 darpa off-line evaluation intrusion detection evaluation. In *Computer Networks*, volume 34. IEEE Computer Society Press, 1996.

[LITH] Lithium, Jesse Ruderman.
http ://www.squarefree.com/lithium/using.html.

[LJ01] Yu Lin and Anita Jones. Application intrusion detection using language library calls. In *Proceedings of the 17th Annual Computer Security Applications Conference (ACSAC 2001)*, December 2001.

[lLWJ98] Jia ling Lin, X. Sean Wang, and Sushil Jajodia. Abstraction-based misuse detection : High-level specifications and adaptable strategies. In *Proceedings of the Eleventh Computer Security Foundations Workshop*, June 1998.

[LMPT98] Ulf Lindqvist, Douglas Moran, Phillip Porras, and Mabry Tyson. Designing idle : The intrusion data library enterprise. Web proceedings of the First International Workshop on Recent Advances in Intrusion Detection (RAID'98), http ://www.raid-symposium.org/raid98, September 1998.

[LSM] Linux Security Module, Linux Kernel.
https ://security.wiki.kernel.org/index.php/Projects.

[LTG+90] Teresa F. Lunt, Ann Tamaru, Fred Gilham, R. Jagannathan, Peter G. Neumann, and Caveh Jalali. Ides : A progress report. In *Proceedings of the Computer Security Application Conference*, pages 273–285, 1990.

[Mad00] Dana Madsen. An operating system analog to the perl data tainting functionality. In *in Proceedings of the 23rd National Information Systems Security Conference*, 2000.

[MFS90] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. In *Communications of the ACM*, volume 33, pages 32–44. ACM, December 1990.

[MG00] Christoph Michael and Anup Ghosh. Using finite automata to mine execution data for intrusion detection : A preliminary report. In H. Debar, L. Mé, and S. F. Wu, editors, *Proceedings of the Third International Workshop on the Recent Advances in Intrusion Detection (RAID'2000)*, number 1907 in LNCS, pages 66–79, October 2000.

[MM88] Aamer Mahmood and Edward J. McCluskey. Concurrent error detection using watchdog processors-a survey. *IEEE Transactions on Computers*, 37(2) :160–174, February 1988.

[MM01] Cédric Michel and Ludovic Mé. ADeLe : an Attack Description Language for Knowledge-based Intrusion Detection. In *Proceedings of the 16th IFIP International Conference on Information Security (IFIP/SEC 2001)*, pages 353–365, June 2001.

[MN95] Lee D. Mcfearin and V.S Sukumaran Nair. Control-flow checking using assertions. In *IFIP Intl Working Conference Dependable Computing for Critical Applications*. Computer Engineering Research Center at the University of Texas at Austin, September 1995.

[MRVK07] Darren Mutz, William Robertson, Giovannni Vigna, and Richard Kemmerer. Exploiting execution context for the detection of anomalous system calls. In *Proceeding of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID'2007)*. Springer, 2007.

[MS08] Benjamin Monate and Julien Signoles. Slicing for security of code. In *Proceedings of the conference on Trusted Computing - Challenges and Applications (Trusted 2008)*, pages 133–142, 2008.

[MTMS07] Frédéric Majorczyk, Eric Totel, Ludovic Mé, and Ayda Saidane. Détection d'intrusions et diagnostic d'anomalies dans un systéme diversifié par comparaison de graphes de flux d'informations. In *Proceedings of the 6th Conference on Security and Network Architectures (SARSSI)*, Annecy, France, June 2007.

[MTMS08] Frédéric Majorczyk, Eric Totel, Ludovic Mé, and Ayda Saidane. Anomaly detection with diagnosis in diversified systems using information flow graphs. In *Proceedings of the 23rd IFIP International Information Security Conference (IFIP SEC 2008)*, pages 301–315, Milano, Italy, September 2008.

[NAC+06] Nuno Neves, Joao Antunes, Miguel Correia, Paulo Verissimo, and Rui Neves. Using attack injection to discover new vulnerabilities. In *International Conference on Dependable Systems and Networks*, pages 457–466, 2006.

[NC03] Caleb C. Noble and Diane J. Cook. Graph-based anomaly detection. In *Proceedings of the ninth international conference on Knowledge discovery and data mining (KDD '03)*, pages 631–636, Washington, D.C, USA, August 2003.

[NS05] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS 2005)*, San Diego, CA, February 2005.

[NTGG+05] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Green, , Jeffrey Shirley, and David Evans. Automatically hardening web applications using precise tainting. In *IFIP Security Conference*, 2005.

[NULLD] Null httpd, Nulllogic.
http ://sourceforge.net/projects/nullhttpd/.

[OGl05] Julien Olivain and Jean Goubault-larrecq. The orchids intrusion detection tool. In *In Kousha Etessami and Sriram Rajamani, editors, Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05), volume 3576 of Lecture Notes in Computer Science*, pages 286–290. Springer, 2005.

[OSSH] OpenSSH, OpenBSD Project - OpenBSD Secure Shell.
http ://frama-c.com/.

[Ove99] Richard E. Overill. Denial of service attacks : Threats and methodologies. *Journal of Financial Crime*, 6(4) :351–354, 1999.

[OWASP] Testing guide appendix c : Fuzz vectors.

[PD00] Jean-Philippe Pouzol and Mireille Ducassé. Handling generic intrusion signatures is not trivial. Extended abstract presented at RAID'2000, October 2000.

[PD01] Jean-Philippe Pouzol and Mireille Ducassé. From Declarative Signatures to Misuse IDS. In W. Lee, L. Mé, and A. Wespi, editors, *Proceedings of the Fourth International Symposium on the Recent Advances in Intrusion Detection (RAID'2001)*, number 2212 in LNCS, pages 1–21, October 2001.

[PEACH] Peach Fuzzing Platform, Michael Eddington (Leviathan Security Group).
http ://peachfuzzer.com/.

[PK02] Korpinen Pekka and Lyytikäinen Kalle. Ssh1 remote root exploit : sshd crc32 compensation attack detector vulnerability explained, 2002.

[PREL] Prelude Hybrid IDS, Yoann Vandoorselaere and PreludeIDS Technologies.
http ://www.prelude-technologies.com/en/welcome/index.html.

[PSJ07] Chetan Parampalli, R. Sekar, and Rob Johnson. A practical mimicry attack against powerful system-call monitors. Technical report, Secure Systems Laboratory, Stony Brook University, 2007.

[PXFZ] ProxyFuzz, The Art Of Fuzzing and SECFORCE.
http ://theartoffuzzing.com/.

[RCK04] Enric Rodríguez-Carbonell and Deepak Kapur. An abstract interpretation approach for automatic generation of polynomial invariants. In *Static Analysis*, volume 3148/2004 of *Lecture Notes in Computer Science*, pages 280–295. Springer Berlin / Heidelberg, 2004.

[RCK05] Enric Rodríguez-Carbonell and Deepak Kapur. Program verification using automatic generation of invariants. In *Theoretical Aspects of Computing - ICTAC 2004*, volume 3407/2005 of *Lecture Notes in Computer Science*, pages 325–340. Springer Berlin / Heidelberg, 2005.

[RCK07] Enric Rodríguez-Carbonell and Deepak Kapur. Automatic generation of polynomial invariants of bounded degree using abstract interpretation. In *Science of Computer Programming*, volume 64, pages 54–75. Elsevier North-Holland, Inc., January 2007.

[Ric53] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. 74 :358–366, 1953.

[ROSM02] Center For Reliable, Nahmsuk Oh, Philip P. Shirvani, and Edward J. Mccluskey. Control-flow checking by software signatures. *IEEE Transactions on Reliability*, 51(111-122), 2002.

[SBDB01] R. Sekar, Mugdha Bendre, Dinakar Dhurjati, and Pradeep Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 144–155, Oakland, CA, May 2001.

[Sch00] Randal L. Schwartz. Taint so easy, is it ? *Sys Admin*, 9(8) :53–54, 2000.

[SCPU06] SPEC CPU2006, Standard Performance Evaluation Corporation.
http ://www.spec.org/cpu2006/, 2006.

[Sec98] Secure Networks. *Custom Attack Simulation Language (CASL)*, January 1998.

[SELIN] Security-Enhanced Linux, USA National Security Agency (NSA).
http ://www.nsa.gov/research/selinux/index.shtml.

[SFUZZ] ShareFuzz, Dave Aitel (Immunity).
http ://sharefuzz.sourceforge.net/.

[SGA07] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing : Brute Force Vulnerability Discovery*. Addison-Wesley Professional, July 2007.

[SM90] Nirmal R. Saxena and Edward J. McCluskey. Control-flow checking using watchdog assists and extended-precision checksums. *IEEE Transactions on Computers*, 39(4) :554–559, April 1990.

[SNARE] Snare, InterSect Alliance.
http ://www.intersectalliance.com/projects/Snare/.

[SPIKE] General Purpose Fuzzing, Dave Aitel (Immunity).
http ://www.immunitysec.com/resources-freesoftware.shtml.

[SQLI] SQL Injection Attacks by Example, Steve Friedl (Unixwiz.net).
http ://unixwiz.net/techtips/sql-injection.html.

[SSMTP] sSMTP - Simple SMTP, Debian.
http ://wiki.debian.org/ssmtp.

[Sta01] Paul Starzetz. Ssh1 crc32 vulnerability analysis, 2001.

[STRAC] Strace, Paul Kranenburg, Dmitry Levin.
http ://sourceforge.net/projects/strace/.

[SWWR04] Robert Stroud, Ian Welch, John Warne, and Peter Ryan. A qualitative analysis of the intrusion-tolerance capabilities of the maftia architecture. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN ?04)*, page 453, 2004.

[SYSA] Syscall Anomaly, The Computer Security Group at UCSB.
http ://www.cs.ucsb.edu/ seclab/projects/libanomaly/.

[TAOF] Taof, The Art Of Fuzzing and SECFORCE.
http ://theartoffuzzing.com/.

[TH05] David Thomas and Andrew Hunt. *Programming Ruby : the pragmatic programmer's guide*. The Pragmatic Programmers, LLC., Raleigh, NC, USA, 2 edition, August 2005.

[Tho07] Yohann Thomas. *Policy-based response to intrusions through context activation*. PhD thesis, Ecole nationale supérieure des télécommunications de Bretagne, 2007.

[Tip95] Frank Tip. A survey of program slicing techniques. Technical report, CWI, 1995.

[TVM04] Eric Totel, Bernard Vivinis, and Ludovic Mé. A Language Driven Intrusion Detection System for Event and Alert Correlation. In *Proceedings ot the 19th IFIP International Information Security Conference*, pages 209–224, Toulouse, August 2004. Kluwer Academic.

[US01] Prem Uppuluri and R. Sekar. Experiences with specification-based intrusion detection. In W. Lee, L. Mé, and A. Wespi, editors, *Proceedings of the Fourth International Symposium on the Recent Advances in Intrusion Detection (RAID'2001)*, number 2212 in LNCS, pages 172–189, Davis, CA, October 2001.

[VEK00] Giovanni Vigna, Steven T. Eckmann, and Richard A. Kemmerer. Attack languages. In *Proceedings of the IEEE Information Survivability Workshop*, October 2000.

[VHM03] Rajesh Venkatasubramanian, John P. Hayes, and Brian T. Murray. Low-cost on-line fault detection using control flow assertions. In *Proceedings of the On-Line Testing Symposium (IOLTS'2003)*, pages 137–143, July 2003.

[VL89] Hank S. Vaccaro and Gunar E. Liepins. Detection of anomalous computer session activity. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 280–289, Oakland, CA, May 1989.

[VNC03] Paulo Esteves Veríssimo, Nuno Ferreira Neves, and Miguel Pupo Correia. Intrusion-tolerant architectures : Concepts and design. In *Architecting Dependable Systems*, volume 2677. Sptringer-Verlag, April 2003.

[WD01] David Wagner and Drew Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001.

[WDD00] Andreas Wespi, Marc Dacier, and Hervé Debar. Intrusion detection using variable-length audit trail patterns. In H. Debar, L. Mé, and S. F. Wu, editors, *Proceedings of the Third International Workshop on the Recent Advances in Intrusion Detection (RAID'2000)*, number 1907 in LNCS, pages 110–129, October 2000.

[Wei81] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[WSB+96] Kenneth M. Walker, Daniel F. Sterne, M. Lee Badger, Michael J. Petkac, David L. Shermann, and Karen A. Oostendorp. Confining root programs with domain and type enforcement (dte). In *Proceedings of the Sixth USENIX UNIX Security Symposium*, July 1996.

[WUFTP] WU-FTPD, Chris Myers and Bryan D. O'Connor. http ://wu-ftpd.therockgarden.ca/.

[WWRS03] Ian Welch, John Wame, Peter Ryan, and Robert Stroud. Architectural analysis of MAFTIAś intrusion tolerance capabilities. Technical Report Deliverable D99, MAFTIA Project, January 2003.

[YWZK09] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSPÓ9, pages 291–304, New York, NY, USA, 2009. ACM.

[Zal01] Michal Zalewski. Ssh crc-32 compensation attack detector vulnerability, 2001.

[ZMB02] Jacob Zimmermann, Ludovic Mé, and Christophe Bidan. Introducing reference flow control for detecting intrusion symptoms at the os level. In Andreas Wespi, Giovanni Vigna, and Luca Deri, editors, *Proceedings*

of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID'2002), volume 2516 of *Lecture Notes in Computer Science*, pages 292–306. Springer, 2002.

[ZMB03a] Jacob Zimmermann, Ludovic Mé, and Christophe Bidan. Experimenting with a policy-based hids based on an information flow control model. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, December 2003.

[ZMB03b] Jacob Zimmermann, Ludovic Mé, and Christophe Bidan. An improved reference flow control model for policy-based intrusion detection. In *Proceedings of the 8th European Symposium on Research in Computer Security (ESORICS)*, October 2003.

Appendices

Annexe A

Comparaison des taux de détection pour n et V quelconques

Afin de généraliser le résultat obtenu à la section 3.1.2.2, il nous faut reprendre le raisonnement non plus sur un exemple donné mais pour des valeurs quelconques de n et de V. Pour cela, nous gardons les notations et les définitions énoncées précédemment puis nous commençons par définir les projections suivantes :

$$\begin{array}{rccc} \pi_i : & D & \rightarrow & D_i \\ & x = (x_1, \ldots, x_n) & \mapsto & \pi_i(x) = x_i \end{array}$$

Cette définition peut-être étendue à un sous-ensemble $A \subseteq D$. On a alors $\pi_i(A) = \{\pi_i(x) \,|\, x \in A\}$. Si on considère $V \subseteq D$, on peut alors définir $\overline{V}$ le produit cartésien des domaines de variation de chacune des variables X_n prises séparément par :

$$\overline{V} = \Pi_{i=1}^{n} \pi_i(V)$$

On sait que pour tout $x = (x_1, \ldots, x_n) \in V$ et pour tout $i \in [1, n]$ on a $\pi_i(x) = x_i \in \pi_i(V)$. On peut donc conclure d'après la définition ci-dessus de $\overline{V}$ que $x \in \overline{V}$. On peut alors en déduire les inclusions ensemblistes suivantes :

$$D \supseteq \overline{V} \supseteq V \supseteq S$$

Soit x un vecteur appartenant à V, l'ensemble des vecteurs corrects obtenus par analyse statique et pour un point donné du programme. Soit $\tilde{x}$ un vecteur obtenue à partir de x en modifiant m de ses composantes. On sait qu'une alerte est levée par notre mécanisme de détection au point d'exécution considéré si et seulement si $\tilde{x} \notin V$. Cette situation se produit dans plusieurs cas pour lesquels nous allons à nouveau séparément comparer les taux de détection pour les injections où $m = 1$ et

$m = 2$. Plus précisément, selon les inclusions ensemblistes de l'équation précédente, ceci ne peut se produire que dans deux cas :

1. Soit $\tilde{x} \in D - \overline{V}$.
2. Soit $\tilde{x} \in \overline{V} - V$.

A.1 Cas où $\tilde{x} \in D - \overline{V}$

Nous commençons par considérer le cas 1. Tout d'abord, nous introduisons pour cela les coefficients μ_i qui représentent les probabilités que le i$^{\text{ème}}$ coefficient de x soit modifié. On a :

$$\sum_{i=1}^{n} \mu_i = 1$$

On note R_1 la probabilité de détection lorsque $\tilde{x}$ est obtenu par modification d'une seule composante de x. Puisque $\tilde{x} \in D - \overline{V}$ et que $x \in V$ et donc $x \in \overline{V}$, on en déduit que :

$$\forall\, i \in [1, n] \quad \pi_i(x) \in \pi_i(V)$$

Comme $\tilde{x}$ est obtenu à partir de x par modification d'une seule de ses composantes, $\tilde{x}$ ne peut être dans $D - \overline{V}$ que s'il existe une unique composante k telle que :

$$\pi_k(\tilde{x}) \notin \pi_k(V)$$

C'est précisément cette composante k qui a été la cible de l'injection. La modification de la variable étant équiprobable sur tout son domaine de définition, la probabilité d'occurrence de cette événement vaut exactement :

$$\frac{\mid D_k \mid - \mid \pi_k(V) \mid}{\mid D_k \mid}$$

Cela représente la probabilité de détection d'une injection qui ne modifie qu'une seule composante de x, sachant que cette composante est la k$^{\text{ème}}$ composante du vecteur. On note cette probabilité r_k et on introduit le coefficient μ_k qui représente la probabilité que la k$^{\text{ème}}$ composante de x soit modifiée. On peut donc en déduire que R_1 vaut :

$$\begin{aligned} R_1 &= \sum_{i=1}^{n} P[\text{modification de la i}^{\text{ème}}\text{ composante}] \times r_i \\ &= \sum_{i=1}^{n} \mu_i r_i \end{aligned}$$

On note maintenant R_2 la probabilité de détection lorsque $\tilde{x}$ est obtenu par modification de deux composantes de x. On note ces deux composantes k et l. En

utilisant le même raisonnement que précédemment, on obtient que la détection ne peut avoir lieu que dans l'un des trois cas disjoints suivants :

$$\begin{array}{c} \pi_k(\tilde{x}) \notin \pi_k(V) \text{ et } \pi_l(\tilde{x}) \in \pi_l(V) \\ \pi_k(\tilde{x}) \in \pi_k(V) \text{ et } \pi_l(\tilde{x}) \notin \pi_l(V) \\ \pi_k(\tilde{x}) \notin \pi_k(V) \text{ et } \pi_l(\tilde{x}) \notin \pi_l(V) \end{array}$$

La probabilité totale d'occurrence de ces trois cas est alors de $r_k(1-r_l)+r_l(1-r_k)+r_k r_l$ soit $r_k + r_l - r_k r_l$. On peut donc en déduire l'expression de R_2 :

$$\begin{aligned} R_2 = & \sum_i \sum_{j \neq i} P[\text{modification de la i}^{\text{ème}} \text{ composante}] \\ & \times P[\text{modification de la j}^{\text{ème}} \text{ composante sachant la i}^{\text{ème}}\text{modifiée}] \\ & \times (r_i + r_j - r_i r_j) \end{aligned}$$

De plus, si μ_i représente la probabilité que la i$^{\text{ème}}$ composante de x soit modifiée, alors la probabilité que la j$^{\text{ème}}$ composante de x soit modifiée connaissant la i$^{\text{ème}}$ composante modifiée est de $\mu_j(1-\mu_i)^{-1}$. On peut donc en déduire que R_2 vaut :

$$R_2 = \sum_i \sum_{j \neq i} \mu_i \frac{\mu_j}{1-\mu_i} (r_i + r_j - r_i r_j)$$

Nous voulons maintenant montrer que le taux de détection d'une injection qui ne modifie qu'une seule variable est toujours inférieur au taux de détection d'une injection qui modifie deux variables. Pour cela, sachant que $\sum_{j \neq i} \mu_j = 1 - \mu_i$ et $\sum_{j \neq i} \mu_j r_j = R_1 - \mu_i r_i$, nous exprimons d'abord R_2 en fonction de R_1 :

$$\begin{aligned} R_2 &= \sum_i \sum_{j \neq i} \mu_i \frac{\mu_j}{1-\mu_i} (r_i + r_j - r_i r_j) \\ &= \sum_i \frac{\mu_i}{1-\mu_i} \left(\sum_{j \neq i} \mu_j r_i + \mu_j r_j - \mu_j r_i r_j \right) \\ &= \sum_i \frac{\mu_i}{1-\mu_i} \left(\sum_{j \neq i} \mu_j r_j + r_i \left(\sum_{j \neq i} \mu_j - \sum_{j \neq i} \mu_j r_j \right) \right) \\ &= \sum_i \frac{\mu_i}{1-\mu_i} \left(R_1 - \mu_i r_i + r_i (1 - \mu_i - R_1 + \mu_i r_i) \right) \\ &= \sum_i \frac{\mu_i}{1-\mu_i} \left(R_1 - \mu_i r_i + r_i - \mu_i r_i - R_1 r_i + \mu_i r_i^2 \right) \\ &= \sum_i \frac{\mu_i}{1-\mu_i} (r_i - \mu_i r_i) + \sum_i \frac{\mu_i}{1-\mu_i} (R_1(1-r_i) + \mu_i r_i (r_i - 1)) \\ &= \sum_i \mu_i r_i + \sum_i \frac{\mu_i}{1-\mu_i} (R_1 - \mu_i r_i)(1 - r_i) \\ &= R_1 + \sum_i \mu_i (1-\mu_i)^{-1} (R_1 - \mu_i r_i)(1 - r_i) \end{aligned}$$

La différence $R_2 - R_1$ peut donc clairement être exprimée par une somme de termes nuls ou positifs :

$$R_2 - R_1 = \sum_i \underbrace{\mu_i(1-\mu_i)^{-1}}_{\geq 0} \underbrace{(R_1 - \mu_i r_i)}_{\geq 0} \underbrace{(1-r_i)}_{\geq 0}$$

Ceci prouve donc bien que dans le cas 1 le taux de détection d'une injection qui modifie deux variables est nécessairement supérieur ou égale au taux de détection d'une injection qui n'en modifie qu'une seule.

A.2 Cas où $\tilde{x} \in \overline{V} - V$

Nous considérons maintenant le cas 2. Soit $\tilde{x_k}(z)$ le vecteur obtenu par le remplacement de la k$^{\text{ième}}$ composante de x par la valeur $z \in D_k$. On a donc $\tilde{x_k}(z) = (x_1, \ldots, x_{k-1}, z, x_{k+1}, \ldots, x_n)$. La probabilité de détection vaut dans ce cas :

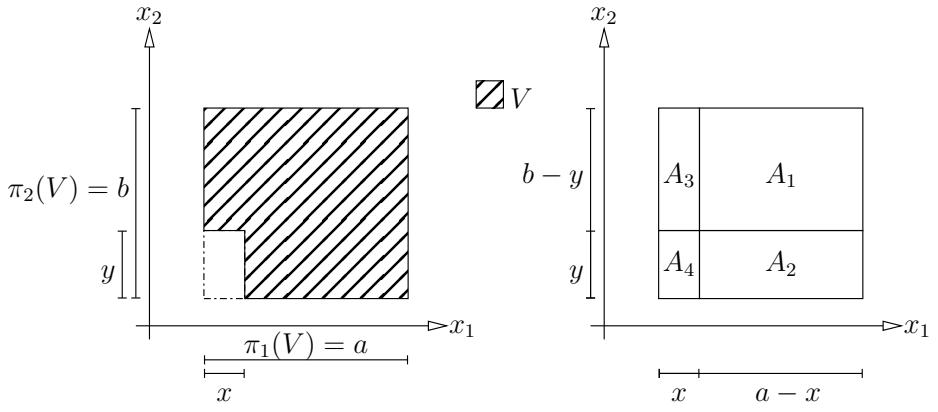
$$\frac{\int_{u \in \pi_k(V)} \mathbb{1}_{\tilde{x}_k(u) \notin V} du}{|\pi_k(V)|}$$

On constate que cette probabilité est une fonction du point x considéré. La probabilité de détection dans ce cas notée $R_1^\star(x_1, \ldots, x_n) = R_1^\star(x)$ vaut donc :

$$R_1^\star(x) = \sum_{k=1}^{n} \frac{\mu_k}{\mid \pi_k(V) \mid} \int_{u \in \pi_k(V)} \mathbb{1}_{\tilde{x}_k(u) \notin V} du$$

On constate maintenant que cette probabilité de détection est là aussi dépendante du point x considéré. Par ailleurs, il est clair que cette définition reste valable pour tout point $x \in \overline{V}$. Pour un point $x \in V \subseteq \overline{V}$, il est évident qu'il s'agit de la probabilité de détection d'une injection lorsqu'une seule variable est affectée. Pour un point $x \in \overline{V} - V$, il s'agit là encore de la probabilité de détection d'une injection lorsqu'une seule variable est affectée, mais en partant d'une valuation se situant hors de la spécification du programme.

Cela peut par exemple se produire dans le cas d'une boucle où une injection a lieu à chaque itération. Lors de la première itération, il est possible que l'injection amène les variables locales à sortir de la spécification du programme. Si une seconde itération peut avoir lieu, une seconde injection peut maintenir les variable en dehors des frontières calculées par l'analyse statique. C'est la probabilité d'occurence de cet évènement que mesure $R_1^\star$ dans ce cas. Rappelons que nous considérons seulement des exécutions dans lesquelles une seule injection est opérée. Donc, même si nous ne considérons pas ce cas en pratique, la définition de $R_1^\star$ permet cependant de le



capturer. On a donc $R_1^\star$ qui est une densité de probabilité définie par :

$$\begin{array}{rccc} R_1^\star : & \overline{V} & \to & [0,1] \\ & x & \mapsto & R_1^\star(x) \end{array}$$

De la même manière que précédemment, nous pouvons aussi considérer la probabilité de détection lorsque $\tilde{x}$ est obtenu par la modification de deux composantes de x. Soient k et l ces deux composantes. On obtient dans ce cas une probabilité de détection égale à :

$$\frac{\displaystyle\int_{u \in \pi_k(V)} \int_{v \in \pi_l(V)} \mathbb{1}_{\tilde{x}_{k,l}(u,v)} du dv}{\mid \pi_k(V) \mid\mid \pi_l(V) \mid}$$

où $\tilde{x}_{k,l}(u,v) = (x_1, \ldots, x_{k-1}, u, x_{k+1}, \ldots, x_{l-1}, v, x_{l+1}, \ldots, x_n)$. On obtient donc une probabilité de détection globale notée $R_2^\star(x_1, \ldots, x_n) = R_2^\star(x)$ égale à :

$$R_2^\star(x) = \sum_k \sum_{l \neq k} \frac{\mu_k \mu_l}{(1-\mu_k) \mid \pi_k(V) \mid\mid \pi_l(V) \mid} \int_{u \in \pi_k(V)} \int_{v \in \pi_l(V)} \mathbb{1}_{\tilde{x}_{k,l}(u,v)} du dv$$

Là aussi $R_2^\star$ dépend du point x considéré. De manière identique à $R_1^\star$, on peut étendre la définition de $R_2^\star$ à tout point $x \in \overline{V}$.

On considère l'exemple illustré par la figure A.1(a). Dans cet exemple $\overline{V}$ est l'hypercube de dimension $a \times b$. Nous allons calculer les valeurs prises par $R_1^\star$ et $R_2^\star$ sur cet exemple. On peut découper $\overline{V}$ en quatre zones distinctes comme illustré par la figure A.1(b). Dans chacune de ces zones $R_1^\star$ est constante. Pour autant, $R_1^\star$ prend des valeurs différentes pour chacune des zones. On a :

$$\begin{cases} \forall\, u \in A_1, & R_1^\star(u) = 0 \\ \forall\, u \in A_2, & R_1^\star(u) = \frac{\mu_1}{a} x + \frac{\mu_2}{b} 0 = \frac{\mu_1 x}{a} \\ \forall\, u \in A_3, & R_1^\star(u) = \frac{\mu_1}{a} 0 + \frac{\mu_2}{b} y = \frac{\mu_2 y}{b} \\ \forall\, u \in A_4, & R_1^\star(u) = \frac{\mu_1}{a} x + \frac{\mu_2}{b} y = \frac{b\mu_1 x + a\mu_2 y}{ab} \end{cases}$$

En ce qui concerne $R_2^\star$, on constate que sa valeur est constante pour tout point $u \in \overline{V}$. On a $R_2^\star(u) = \mu_1 \frac{\mu_2}{1-\mu_1} \frac{1}{ab} xy + \mu_2 \frac{\mu_1}{1-\mu_2} \frac{1}{ab} xy = \frac{xy}{ab}$. On peut donc constater que pour la zone A_1, on a $R_1^\star(u) \leq R_2^\star(u)$, alors que pour la zone A_2 si $\mu_1 \geq \frac{y}{b}$ (ce qui est parfaitement possible), on aura $R_1^\star(u) \geq R_2^\star(u)$. Ce contre-exemple montre clairement qu'il est inutile de chercher à prouver que $\forall\, u \in V \quad R_1^\star(u) \leq R_2^\star(u)$. On va donc plutôt chercher à montrer que cette inégalité est vraie seulement en moyenne. Pour cela, on introduit la notation suivante :

$$\overline{F}_{|_X} = \frac{1}{\mid X \mid} \int_{x \in X} F(x) dx.$$

On va maintenant calculer $\overline{R_1^\star}_{|_V}, \overline{R_1^\star}_{|_{\overline{V}}}, \overline{R_2^\star}_{|_V}$ et $\overline{R_2^\star}_{|_{\overline{V}}}$. Comme $R_2^\star$ est une constante sur $\overline{V}$, on a de manière naturelle $\overline{R_2^\star}_{|_V} = \overline{R_2^\star}_{|_{\overline{V}}} = \frac{xy}{ab}$. En ce qui concerne $R_1^\star$, on a :

$$\begin{aligned}
\overline{R_1^\star}_{|_{\overline{V}}} &= \frac{1}{\mid \overline{V} \mid} \int_{x \in \overline{V}} R_1^\star(x) dx \\
&= \frac{1}{ab} \left(\int_{x \in A_1} R_1^\star(x) dx + \int_{x \in A_2} R_1^\star(x) dx + \int_{x \in A_3} R_1^\star(x) dx + \int_{x \in A_4} R_1^\star(x) dx \right) \\
&= \tfrac{1}{ab} \left(\mid A_1 \mid .0 + \mid A_2 \mid . \tfrac{\mu_1 x}{a} + \mid A_3 \mid . \tfrac{\mu_2 y}{b} + \mid A_4 \mid . \tfrac{b\mu_1 x + a\mu_2 y}{ab} \right) \\
&= \tfrac{1}{ab} \left(y(a-x) \tfrac{\mu_1 x}{a} + x(b-y) \tfrac{\mu_2 y}{b} + xy \tfrac{b\mu_1 x + a\mu_2 y}{ab} \right) \\
&= \tfrac{1}{(ab)^2} \left(by(a-x)\mu_1 x + ax(b-y)\mu_2 y + x^2 y b \mu_1 + xy^2 a \mu_2 \right) \\
&= \tfrac{1}{(ab)^2} \left(abxy\mu_1 - x^2 by\mu_1 + abxy\mu_2 - axy^2\mu_2 + x^2 yb\mu_1 + xy^2 a\mu_2 \right) \\
&= \tfrac{1}{(ab)^2} \left(abxy(\mu_1 + \mu_2) + xy(bx\mu_1 + ay\mu_2 - bx\mu_1 - ay\mu_2) \right) \\
&= \tfrac{1}{(ab)^2} abxy = \tfrac{xy}{ab} = \overline{R_2^\star}_{|_{\overline{V}}}
\end{aligned}$$

et

$$\begin{aligned}
\overline{R_1^\star}_{|_V} &= \frac{1}{\mid V \mid} \int_{x \in V} R_1^\star(x) dx \\
&= \frac{1}{ab} \left(\int_{x \in A_1} R_1^\star(x) dx + \int_{x \in A_2} R_1^\star(x) dx + \int_{x \in A_3} R_1^\star(x) dx \right) \\
&= \tfrac{1}{ab - xy} \left(\mid A_1 \mid .0 + \mid A_2 \mid . \tfrac{\mu_1 x}{a} + \mid A_3 \mid . \tfrac{\mu_2 y}{b} \right) \\
&= \tfrac{1}{ab - xy} \left(y(a-x) \tfrac{\mu_1 x}{a} + x(b-y) \tfrac{\mu_2 y}{b} \right) \\
&= \tfrac{1}{ab(ab-xy)} \left(abxy\mu_1 - bx^2 y\mu_1 + abxy\mu_2 - axy^2\mu_2 \right) \\
&= \tfrac{1}{ab(ab-xy)} \left(abxy - xy(bx\mu_1 + ay\mu_2) \right) \\
&= \tfrac{xy}{ab(ab-xy)} \left(ab - bx\mu_1 - ay\mu_2 \right).
\end{aligned}$$

On peut donc vérifier que $\overline{R_{1|_V}^\star} \leq \overline{R_{2|_V}^\star}$:

$$\begin{aligned}
\overline{R_{1|_V}^\star} - \overline{R_{2|_V}^\star} &= \tfrac{xy}{ab(ab-xy)}\left(ab - bx\mu_1 - ay\mu_2\right) - \tfrac{xy}{ab} \\
&= \tfrac{xy}{ab(ab-xy)}\left(ab - bx\mu_1 - ay\mu_2 - (ab - xy)\right) \\
&= \tfrac{xy}{ab(ab-xy)}\left(xy - bx\mu_1 - ay\mu_2\right) \\
&= \tfrac{xy}{ab(ab-xy)}\left(\mu_1 xy - bx\mu_1 + \mu_2 xy - \mu_2 ay\right) \\
&= \underbrace{\frac{xy}{ab(ab-xy)}}_{\geq 0}\left(\underbrace{\mu_1 x}_{\geq 0}\underbrace{(y-b)}_{\leq 0} + \underbrace{\mu_2 y}_{\geq 0}\underbrace{(x-a)}_{\leq 0}\right) \\
&\leq 0.
\end{aligned}$$

Nous allons donc prouver ces deux propriétés de manière générale. À savoir que :

$$\begin{cases} \overline{R_{1|_{\overline{V}}}^\star} = \overline{R_{2|_{\overline{V}}}^\star} = 1 - \frac{|V|}{|\overline{V}|} \\ \overline{R_{1|_V}^\star} \leq \overline{R_{2|_V}^\star} \end{cases}$$

Nous commençons par prouver la première propriété. On a :

$$\begin{aligned}
\overline{R_{1|_{\overline{V}}}^\star} &= \frac{1}{|\,\overline{V}\,|}\int_{x\in\overline{V}} R_1^\star(x)dx \\
&= \frac{1}{|\,\overline{V}\,|}\int_{x\in\overline{V}} \sum_{k=1}^{n} \frac{\mu_k}{|\,\pi_k(V)\,|}\int_{y\in\pi_k(V)} \mathbb{1}_{\tilde{x}_k(y)\notin V} dydx \\
&= \frac{1}{|\,\overline{V}\,|}\sum_{k=1}^{n} \frac{\mu_k}{|\,\pi_k(V)\,|}\int_{x\in\overline{V}}\int_{y\in\pi_k(V)} \mathbb{1}_{\tilde{x}_k(y)\notin V} dydx \\
&= \frac{1}{|\,\overline{V}\,|}\sum_{k=1}^{n} \frac{\mu_k}{|\,\pi_k(V)\,|}\int_{x_1\in\pi_1(V)}\dots\int_{x_n\in\pi_k(V)}\int_{y\in\pi_k(V)} \mathbb{1}_{\tilde{x}_k(y)\notin V} dydx_n\dots dx_1 \\
&= \frac{1}{|\,\overline{V}\,|}\sum_{k=1}^{n} \frac{\mu_k}{|\,\pi_k(V)\,|}\int_{x_k\in\pi_1(V)} \underbrace{\int\dots\int_{\Pi_{j\neq k}x_j\in\pi_j(V)}\int_{y\in\pi_k(V)}}_{(x_1,\dots,x_{k-1},y,x_{k+1},\dots,x_n)=\tilde{x}_k(y)\text{ décrit }\overline{V}} \mathbb{1}_{\tilde{x}_k(y)\notin V} dydx_n\dots dx_1dx_k \\
&= \frac{1}{|\,\overline{V}\,|}\sum_{k=1}^{n} \frac{\mu_k}{|\,\pi_k(V)\,|}\int_{x_k\in\pi_k(V)}\int_{x\in\overline{V}} \mathbb{1}_{x\notin V} dxdx_k \\
&= \frac{1}{|\,\overline{V}\,|}\sum_{k=1}^{n} \frac{\mu_k}{|\,\pi_k(V)\,|}\int_{x_k\in\pi_k(V)} \left(|\,\overline{V}\,| - |\,V\,|\right) dx_k \\
&= \frac{1}{|\,\overline{V}\,|}\sum_{k=1}^{n} |\,\pi_k(V)\,|\left(|\,\overline{V}\,| - |\,V\,|\right) \\
&= \frac{1}{|\,\overline{V}\,|}\sum_{k=1}^{n} \mu_k\left(|\,\overline{V}\,| - |\,V\,|\right) \\
&= \frac{1}{|\,\overline{V}\,|}\left(|\,\overline{V}\,| - |\,V\,|\right)\sum_{k=1}^{n}\mu_k
\end{aligned}$$

$$= \frac{|\overline{V}| - |V|}{|\overline{V}|}$$
$$= 1 - \frac{|V|}{|\overline{V}|}$$

Par les mêmes transformations, on peut montrer que $\overline{R_2^\star}_{|\overline{V}} = 1 - \frac{|V|}{|\overline{V}|}$. Ce qui prouve la première propriété. Pour prouver la seconde propriété, on commence par procéder à des transformations algébrique sur $R_2^\star(x)$:

$$\begin{aligned}
R_2^\star(x) =& \sum_k \sum_k \frac{\mu_k}{(1-\mu_k) \mid \pi_k(V) \mid} \sum_{l \neq k} \frac{\mu_l}{\mid \pi_l(V) \mid} \int_{u \in \pi_k(V)} \int_{v \in \pi_l(V)} \mathbb{1}_{\tilde{x}_{k,l}(u,v)} du dv \\
=& \sum_k \sum_k \frac{\mu_k}{(1-\mu_k) \mid \pi_k(V) \mid} \int_{u \in \pi_k(V)} \left(\underbrace{\sum_{l \neq k} \frac{\mu_l}{\mid \pi_l(V) \mid} \int_{v \in \pi_l(V)} \mathbb{1}_{\tilde{x}_{k,l}(u,v)} dv}_{R_1^\star(x_1,\ldots,x_{k-1},u,x_{k+1},\ldots,x_n) - \frac{\mu_k}{|\pi_k(V)|} \int_{v \in \pi_k(V)} \mathbb{1}_{\tilde{x}_k(v)} dv} \right) du \\
=& \sum_k \sum_k \frac{\mu_k}{(1-\mu_k) \mid \pi_k(V) \mid} \left(\int_{u \in \pi_k(V)} R_1^\star(\tilde{x}_k(u)) du - \int_{u \in \pi_k(V)} \frac{\mu_k}{\mid \pi_k(V) \mid} \int_{v \in \pi_k(V)} \mathbb{1}_{\tilde{x}_k(v)} dv du \right) \\
=& \sum_k \sum_k \frac{\mu_k}{(1-\mu_k) \mid \pi_k(V) \mid} \left(\int_{u \in \pi_k(V)} R_1^\star(\tilde{x}_k(u)) du - \mid \pi_k(V) \mid \frac{\mu_k}{\mid \pi_k(V) \mid} \int_{v \in \pi_k(V)} \mathbb{1}_{\tilde{x}_k(v)} dv \right) \\
=& \sum_k \sum_k \frac{\mu_k}{(1-\mu_k) \mid \pi_k(V) \mid} \left(\int_{u \in \pi_k(V)} R_1^\star(\tilde{x}_k(u)) du - \mu_k \int_{u \in \pi_k(V)} \mathbb{1}_{\tilde{x}_k(u)} du \right)
\end{aligned}$$

On peut alors calculer la différence $\Delta = \overline{R_1^\star}_{|V} - \overline{R_2^\star}_{|V}$:

$$\begin{aligned}
\Delta =& \frac{1}{\mid V \mid} \int_{x \in V} R_1^\star(x) - R_2^\star(x) dx \\
=& \frac{1}{\mid V \mid} \int_{x \in V} \left(\sum_{k=1}^n \frac{\mu}{\mid \pi_k(V) \mid} \int_{y \in \pi_k(V)} \mathbb{1}_{\tilde{x}_k(y) \notin V} dy - \sum_{k=1}^n \frac{\mu_k}{(1-\mu_k) \mid \pi_k(V) \mid} \left(\int_{y \in \pi_k(V)} R_1^\star(\tilde{x}_k(y)) \mu_k \mathbb{1}_{\tilde{x}_k(y) \notin V} dy \right) \right) dx \\
=& \frac{1}{\mid V \mid} \int_{x \in V} \left(\sum_{k=1}^n \frac{\mu_k}{(1-\mu_k) \mid \pi_k(V) \mid} \int_{y \in \pi_k(V)} (1-\mu_k) \mathbb{1}_{\tilde{x}_k(y) \notin V} - R_1^\star(\tilde{x}_k(y)) + \mu_k \mathbb{1}_{\tilde{x}_k(y) \notin V} dy \right) dx \\
=& \frac{1}{\mid V \mid} \int_{x \in V} \left(\sum_{k=1}^n \frac{\mu_k}{(1-\mu_k) \mid \pi_k(V) \mid} \int_{y \in \pi_k(V)} \left(\mathbb{1}_{\tilde{x}_k(y) \notin V} - R_1^\star(\tilde{x}_k(y)) \right) dy \right) dx \\
=& \frac{1}{\mid V \mid} \sum_{k=1}^n \frac{\mu_k}{(1-\mu_k) \mid \pi_k(V) \mid} \int_{x \in V} \int_{y \in \pi_k(V)} \left(\mathbb{1}_{\tilde{x}_k(y) \notin V} - R_1^\star(\tilde{x}_k(y)) \right) dy dx \\
=& \frac{1}{\mid V \mid} \sum_{k=1}^n \frac{\mu_k}{(1-\mu_k) \mid \pi_k(V) \mid} \int_{x \in \overline{V}} \mathbb{1}_{x \in V} \left(\int_{y \in \pi_k(V)} \left(\mathbb{1}_{\tilde{x}_k(y) \notin V} - R_1^\star(\tilde{x}_k(y)) \right) dy \right) dx \\
\leq& \frac{1}{\mid V \mid} \sum_{k=1}^n \frac{\mu_k}{(1-\mu_k) \mid \pi_k(V) \mid} \int_{x \in \overline{V}} \int_{y \in \pi_k(V)} \left(\mathbb{1}_{\tilde{x}_k(y) \notin V} - R_1^\star(\tilde{x}_k(y)) \right) dy dx \\
\leq& \frac{1}{\mid V \mid} \sum_{k=1}^n \frac{\mu_k}{(1-\mu_k) \mid \pi_k(V) \mid} \int \cdots \int_{(x_1,\ldots,x_n) \in \Pi_{j=1}^k \pi_j(V)} \int_{y \in \pi_k(V)} \left(\mathbb{1}_{\tilde{x}_k(y) \notin V} - R_1^\star(\tilde{x}_k(y)) \right) dy dx_n \ldots dx_1 \\
\leq& \frac{1}{\mid V \mid} \sum_{k=1}^n \frac{\mu_k}{(1-\mu_k) \mid \pi_k(V) \mid} \int_{x_k \in \pi_k(V)} \underbrace{\int \cdots \int_{\Pi_{j \neq k} x_j \in \pi_j(V)} \int_{y \in \pi_k(V)}}_{(x_1,\ldots,x_{k-1},y,x_{k+1},\ldots,x_n) = \tilde{x}_k(y) \text{ décrit } \overline{V}} \left(\mathbb{1}_{\tilde{x}_k(y) \notin V} - R_1^\star(\tilde{x}_k(y)) \right) dy dx_n \ldots dx_1
\end{aligned}$$

$$
\begin{aligned}
&\leq \frac{1}{\mid V \mid} \sum_{k=1}^{n} \frac{\mu_k}{(1-\mu_k) \mid \pi_k(V) \mid} \int_{x_k \in \pi_k(V)} \int_{x \in \overline{V}} (\mathbb{1}_{x \notin V} - R_1^\star(x))\, dx dx_k \\
&\leq \frac{1}{\mid V \mid} \sum_{k=1}^{n} \frac{\mu_k}{(1-\mu_k) \mid \pi_k(V) \mid} \mid \pi_k(V) \mid \left(\int_{x \in \overline{V}} \mathbb{1}_{x \notin V} dy - \int_{x \in \overline{V}} R_1^\star(x) dx \right) \\
&\leq \frac{1}{\mid V \mid} \sum_{k=1}^{n} \frac{\mu_k}{(1-\mu_k)} \left(\mid \overline{V} \mid - \mid V \mid - \mid \overline{V} \mid \overline{R_1^\star}_{\mid \overline{V}} \right) \\
&\leq \frac{1}{\mid V \mid} \sum_{k=1}^{n} \frac{\mu_k}{(1-\mu_k)} \left(\mid \overline{V} \mid - \mid V \mid - (\mid \overline{V} \mid - \mid V \mid) \right) \\
&\leq 0
\end{aligned}
$$

Ces résultats prouvent donc que dans le cas 2 le taux de détection d'une injection qui modifie deux variables est nécessairement supérieur ou égale au taux de détection d'une injection qui n'en modifie qu'une seule. Au final, nous avons donc montré que pour n et V quelconques, et ce quelque soit le cas dans lequel on se trouve, en augmentant le nombre de variables modifiées de 1 à 2 on augmente également le taux de détection.

VU :
Le Directeur de Thèse

VU :
Le Responsable de l'École Doctorale

VU pour autorisation de soutenance
Rennes, le

Le Président de l'Université de Rennes 1

Guy CATHELINEAU

VU après soutenance pour autorisation de publication :
Le Président du Jury,