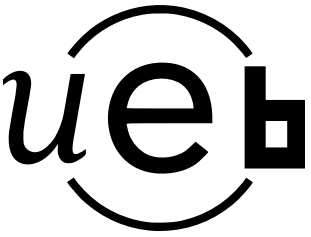
Techniques de détection d'erreur appliquées à la détection d'intrusion

Eric Totel

ANNÉE 2012



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

pour l'obtention de
L'HABILITATION A DIRIGER DES RECHERCHES

Mention : Informatique
présentée par

Eric TOTEL

préparée à l'unité de recherche EPC CIDRE - INRIA/Supelec
INRIA - Ecole Supérieure d'électricité

Techniques de détection d'erreur appliquées à la détection d'intrusion

HDR soutenue à Rennes
le 6 décembre 2012

devant le jury composé de :

Pr. MICHEL CUKIER
Professeur de l'université du Maryland / *Rapporteur*
Pr. RADU STATE
Professeur de l'université du Luxembourg / *Rapporteur*
Dr. OLIVIER FESTOR
Directeur de Recherche INRIA / *Rapporteur*
Pr. SANDRINE BLAZY
Professeur à l'université de Rennes 1 / *Examinatrice*
Pr. HERVÉ DEBAR
Professeur à Telecom Sud-Paris / *Examinateur*
Pr. LUDOVIC MÉ
Professeur à Supelec / *Examinateur*

Remerciements

Je tiens à remercier tous les membres de l'équipe SSIR (devenue CIDRE) pour ces dix années à travailler dans la bonne humeur et tout particulièrement Ludovic Mé pour m'avoir fait découvrir le domaine de la détection d'intrusion.

Je tiens également à remercier Michel Hurfin, Frédéric Tronel, Nicolas Prigent et Christophe Bidan pour leur lecture attentive de ce manuscrit.

Mes remerciements vont également vers les rapporteurs Michel Cukier, Radu State et Olivier Festor pour avoir accepté de rapporter cette HDR.

Enfin je tiens à remercier les doctorants ou ingénieurs de recherche avec qui j'ai travaillé et qui ont rendu possibles les travaux que je mets en valeur dans ce document. Je citerai en particulier Frédéric Majorczyk, Jonathan-Christopher Demay, Olivier Sarrouy et Romaric Ludinard.

Table des matières

Présentation du document

Ce document constitue un dossier de demande d'inscription à l'Habilitation à Diriger des Recherches. Il résume 10 années d'activités professionnelles passées en tant qu'enseignant-chercheur sur le campus de Rennes de Supélec.

Ce document est constitué de deux parties.

La première partie propose une présentation du candidat qui prend la forme d'un curriculum vitæ, d'une présentation des activités d'enseignement et d'une présentation des activités de recherche. L'ensemble se termine par une liste de publications.

La seconde partie est une synthèse d'une partie des activités de recherche menées ces dix dernières années. Un état de l'art pose les concepts sur lesquels reposent ces travaux. Ensuite quatre activités de recherche sont présentées, chacune d'elle mettant en évidence la pertinence de certaines techniques de détection d'erreur dans le domaine de la détection d'intrusion.

Première partie

Présentation du candidat

Curriculum Vitae

Etat civil

Eric Totel. Nationalité française. Né le 27 mai 1970 à Troyes. Célibataire.
Adresse personnelle : 2 rue Molière, Rennes.

Situation Professionnelle

Depuis septembre 2002, enseignant-chercheur à SUPELEC.
Adresse professionnelle : Avenue de la Boulaie, CS 47601, 35576 Cesson Sévigné Cedex.
Tél : 02.99.84.45.00. Mél : `Eric.Totel@supelec.fr`

Discipline de recherche

Informatique ; sûreté de fonctionnement ; sécurité des systèmes d'information ; détection d'intrusion.

Titres universitaires

1998 : Doctorat, Informatique, Institut National Polytechnique de Toulouse, mention très honorable, 1998.
1994 : Diplôme d'études approfondies (DEA) d'informatique, Institut National Polytechnique de Toulouse.
1994 : Diplôme d'ingénieur ENSEEIHT (Ecole Nationale Supérieure d'Electronique, d'Electrotechnique, d'Informatique, d'Hydraulique et des Télécommunications).

Société savante

Membre du comité de pilotage du club 63 (Système Informatique de Confiance) de la SEE (Société de l'Electricité, de l'Electronique et des Technologies de l'Information et de la Communication).

Chapitre 2

Activités d'enseignement

2.1 Introduction

Avant de présenter mes activités d'enseignement, il me semble essentiel d'exposer le cheminement tant personnel que professionnel qui m'a conduit au poste d'enseignant-chercheur que j'occupe aujourd'hui à Supelec.

J'ai commencé à enseigner durant ma première année de thèse, en 1995. À l'époque, il s'agissait d'assurer des vacations à l'IUT de Blagnac dans le domaine de la programmation. Cette première expérience m'a permis de m'apercevoir que j'aimais ce type d'activité, au point de vouloir en faire mon métier. C'est la raison pour laquelle j'ai continué à faire des vacations pendant les quatre années durant lesquelles j'ai travaillé dans l'industrie. Ces vacations ont été données dans des écoles d'ingénieur de Toulouse telles que l'ENSEEIHT (programmation CORBA) ou l'INSA de Toulouse (programmation en ADA) par exemple.

Outre l'intérêt pour les activités de recherche qu'on me proposait, ce goût pour l'enseignement a été déterminant dans mon choix de rejoindre Supelec en 2002.

Ce chapitre présente un récapitulatif de l'ensemble des enseignements que j'ai réalisés depuis mon arrivée à Supelec en 2002, et n'inclut donc pas les vacations effectuées avant cette date. Ces enseignements sont présentés dans l'ordre suivant : les enseignements magistraux, les travaux dirigés, les travaux de laboratoire et enfin les encadrements de projets étudiants. Pour chacune de ces catégories, les enseignements sont classés suivant le niveau où ils ont été dispensés (1ère année, 2ème année ou 3ème année d'école d'ingénieur, et mastères spécialisés). Les charges d'enseignement sont données en heures réelles, à l'exception du tableau récapitulatif de charge en section 2.7 où les chiffres sont donnés en heures équivalentes TD.

2.2 Enseignement magistral

Les cours que j'ai dispensés ces dix dernières années portent sur mes centres d'intérêts en informatique : les langages de programmation et leur apprentissage d'un coté, et la sûreté de fonctionnement de l'autre. Ces deux facettes de mes activités d'enseigne-

ment ont été déclinées dans la plupart des options dans lesquelles il m'a été donné d'enseigner. Les majeures de 3ème année auxquelles j'ai participé sont ISR (Informatique Systèmes et Réseaux), SIS (Systèmes d'information Sécurisé), ISA (Ingénierie des Systèmes Automatisés). Les mastères spécialisés sont RIT (Réseaux Informatique et Télécommunications) devenu ARC (Architecte de Réseaux de Communication), SSI (Sécurité des Systèmes d'information) devenu le mastère CS (Cyber-Sécurité). En outre, en parallèle de la deuxième année d'école d'ingénieur, il existe une filière d'intégration d'étudiants étrangers dans le cursus ingénieur nommée N+I.

2.2.1 Enseignements magistraux dispensés à Supelec en 1ère année

Années	Intitulé	Charge
2010-2011	Cours Modèles de programmation	10h30
2011-2012	Cours Modèles et langages de programmation	10h30

2.2.2 Enseignements magistraux dispensés à Supelec en 2ème année

Années	Intitulé	Charge
2005-2006 à 2011-2012	Cours Langage C	3h

2.2.3 Enseignements magistraux dispensés à Supelec en 3ème année (option ISA, options ISR puis SIS)

Années	Intitulé	Charge
2004-2005 à 2011-2012	Cours Langage C (ISR)	3h
2004-2005 à 2006-2007	Cours Sûreté de fonctionnement (ISR)	3h
2005-2006 à 2011-2012	Cours Sûreté de fonctionnement (ISA)	4,5h
2007-2008 à 2011-2012	Cours Disponibilité (ISR/SIS et M2R université)	3h
2009-2010 à 2011-2012	Cours Langage C++	3h

2.2.4 Enseignements magistraux dispensés à Supelec en mastère RIT puis ARC

Années	Intitulé	Charge
2007-2008 à 2010-2011	Cours Langage C	3h
2008-2009 à 2010-2011	Cours SDL	3h
2008-2009 à 2010-2011	Cours Disponibilité	3h

2.2.5 Enseignements magistraux à Supelec en mastère SSI puis CS

Années	Intitulé	Charge
2002-2003 à 2007-2008	Cours Sûreté de fonctionnement	9h
2003-2004 à 2011-2012	Cours Langage C	4h30

2.3 Travaux dirigés

2.3.1 Travaux dirigés encadrés à Supelec en 1ère année

Années	Intitulé	Charge
2002-2003 à 2009-2011	TDs Modèles de programmation	3h
2002-2003 à 2009-2010	TDs Fondements de l'informatique	3h
2010-2011	TDs Fondements de l'informatique	4h30
2011-2012	TDs Fondements de l'informatique	6h00

2.3.2 Travaux dirigés encadrés à Supelec en 2ème année

Années	Intitulé	Charge
2002-2003 à 2011-2012	TDs Architectures des systèmes informatique	6h

2.3.3 Travaux dirigés encadrés à Supelec en 3ème année (options ISR puis SIS)

Années	Intitulé	Charge
2003-2004 à 2007-2008	TD Glade	1h30
2004-2005 à 2009-2010	TD Systèmes distribués	1h30
2008-2009 à 2009-2010	TD Lex et Yacc	1h30
2010-2011 à 2011-2012	TD Disponibilité	1h30

2.3.4 Travaux dirigés encadrés à Supelec en mastère MERIT puis ARC

Années	Intitulé	Charge
2006-2007 à 2010-2011	TD Sûreté de fonctionnement	1h30

2.4 Travaux de laboratoires

A Supelec, les travaux de laboratoires peuvent être classés en deux catégories distinctes : les *BE* (Bureaux d'Études) d'un côté qui sont des travaux pratiques très dirigés par l'enseignant, et les *TL* (Travaux de Laboratoire), où les étudiants doivent par eux-mêmes réaliser le sujet qui leur est proposé, avec une aide plus légère de l'enseignant. Les BE durent deux créneaux de 1h30, alors que les TL prennent trois créneaux.

2.4.1 Travaux de laboratoires encadrés à Supelec en 1ère année

Années	Intitulé	Charge
2006-2007 à 2010-2011	TLs Modèles de programmation	18h
2006-2007 à 2010-2011	TLs Fondements de l'informatique	24h
2010-2011 à 2011-2012	BEs Modèles et langages de programmation	9h00
2010-2011 à 2011-2012	TLs Modèles et langages de programmation	9h00

2.4.2 Travaux de laboratoires encadrés à Supelec en 2ème année et N+I

Années	Intitulé	Charge
2002-2003 à 2011-2012	TLs Architecture des systèmes informatiques	36h

2.4.3 Travaux de laboratoires encadrés à Supelec en 3ème année (options ISR puis SIS)

Années	Intitulé	Charge
2002-2003 à 2007-2008	TLs Multi agents	22h30
2004-2005 à 2011-2012	BEs Langage C	3h
2004-2005 à 2007-2008	TLs SDL	27h
2005-2006 à 2006-2007	BEs CVS	3h

2.4.4 Travaux de laboratoires encadrés à Supelec en mastère MERIT puis ARC

Années	Intitulé	Charge
2003-2004 à 2010-2011	BE SDL Bit Alterné	3h
2008-2009 à 2010-2011	BE Langage C	3h
2008-2009 à 2009-2010	BE Multithreading	3h
2010-2011	BEs Multithreading	6h

2.4.5 Travaux de laboratoires encadrés en mastère SSI devenu CS

Années	Intitulé	Charge
2003-2004 à 2009-2010	BEs Langage C	6h
2010-2011 à 2011-2012	BEs Langage C	9h

2.5 Encadrement de projets

Mon activité d'encadrement de projets depuis 2002 se décline en deux parties distinctes : d'un côté les projets logiciels de deuxième année, de l'autre les projets de troisième année ou de mastère spécialisé.

Les projets logiciel de deuxième année permettent aux étudiants de réaliser pour la première fois un logiciel. Ces projets sont choisis pour leur intérêt pédagogique et ne présentent généralement pas de difficulté majeure. Toutefois, ils permettent aux étudiants d'acquérir une première expérience en programmation, la plupart du temps en Java.

Les projets de troisième année, au contraire, doivent fournir aux étudiants un défi important, les mettant en situation réelle de réalisation d'un projet en entreprise. D'ailleurs nombre de ces projets sont réalisés en partenariat avec un industriel, dans le cadre de CEI (Contrat d'Étude Industrielle). L'industriel fournit aux étudiants un

sujet d'étude, et participe avec les enseignants à l'encadrement du projet. L'industriel, dans le cadre du CEI, se présente comme un client du point de vue des étudiants.

2.5.1 Projets logiciels en 2ème année

Il faut noter ici que certains sujets reviennent parfois plusieurs fois au travers des années, soit parce qu'ils ont mené à des résultats décevants, ou qu'ils ont montré un intérêt pédagogique particulier.

Années	Intitulé du projet
2002-2003	Architecture pour la détection d'intrusion Réalisation d'une interface graphique visualisant des alertes IDMEF
2003-2004	Bataille navale Logiciel de dessin de fractales en 2D Génération de paysages fractales en 3D
2004-2005	BibTeX pour Word BibTeX pour OpenOffice Visualisation d'une architecture réseau Serveurs FTP diversifiés (2 groupes)
2005-2006	Gestion et visualisation d'une base de données cinématographique BibTeX pour Word Logiciel de généalogie
2006-2007	Logiciel de généalogie Logiciel de dictée
2007-2008	Logiciel de généalogie Statistiques sur une base de données bibliographique Base de données cinématographique Logiciel de dictée
2008-2009	Logiciel de gestion de commandes Génération automatique de labyrinthe Visualisation d'un labyrinthe en 3 dimensions
2009-2010	Carnet de rendez-vous partagé Gestion et affichage d'une tablature
2010-2011	Affichage d'un labyrinthe en 3D Logiciel de dictée Détection et suivi de personnes
2011-2012	Gestion des évènements sportifs organisés par le BDS Génération de dessin à base de post-it

2.5.2 Projets de 3ème année et mastère spécialisé

Années	Type	Intitulé du projet
2003-2004	CEI	Etude du piratage de films sur internet
2004-2005	Projet 3A	Interception d'appels système sous Windows
	Projet mastère	Construction d'une base de signatures d'attaques pour GnG
2005-2006	CEI	Étude de la sécurité et disponibilité d'un réseau sans fil embarqué
	Projet 3A	Mise en place d'une plateforme d'attaque/défense
2006-2007	Projet 3A	Réalisation d'un IDS fondé sur la diversification fonctionnelle
2007-2008	Projet 3A	Sécurisation d'un outil de bibliographie multi-utilisateurs
2008-2009	Projet 3A	Projet sécurité CORBA
2009-2010	CEI	Solutions de réplication de bases de données PostgreSQL
2010-2011	Projet 3A	Évaluation d'un instrumenteur de code sur des exemples choisis
2011-2012	Projet mastère	Réalisation de la phase d'apprentissage pour un système de détection d'intrusion dans les applications Ruby on Rails

2.6 Formation continue

La formation continue, troisième mission de base des enseignants-chercheurs de Supelec, est, par nature, soumise aux variations du marché et aux évolutions constantes des besoins des entreprises clientes. En ce qui me concerne, les activités de formation continue que j'ai menées depuis 2002 sont peu nombreuses. Ces interventions ont toutes tourné autour de la sécurité informatique.

2.7 Bilan de la charge d'enseignement à Supelec

Ce dernier tableau présente un résumé de ma charge d'enseignement en heures équivalentes TD depuis 2002, telle que fournie par la direction des études. Exception faite des deux premières années, cette charge dépasse généralement la centaine d'heures. J'ai également participé à des enseignements dans d'autres établissements, mais cela reste infime comparé au nombre d'heures réalisées à Supelec.

Années	Heures équivalentes TD
2002-2003	59
2003-2004	74
2004-2005	127
2005-2006	130
2006-2007	121
2007-2008	122
2008-2009	92
2009-2010	119
2010-2011	109

Chapitre 3

Activités de recherche

3.1 Contexte de la recherche

Alors que j'étais en étudiant en dernière année en école d'ingénieur, j'ai eu l'opportunité de réaliser mon stage de fin d'étude au LAAS-CNRS dans l'équipe TSF (Tolérance aux fautes et sûreté de fonctionnement). Le sujet portait sur l'implémentation dans un environnement Unix de mécanismes de tolérances aux fautes dans des systèmes distribués : la fragmentation-redondance-dissémination (*FRD*). Cette approche m'a permis de découvrir l'existence de mécanismes unifiés permettant à la fois de tolérer des fautes accidentelles et des fautes intentionnelles dans les systèmes. L'intérêt que je portais au sujet m'a amené à poursuivre en thèse dans la même équipe.

La thèse que j'ai réalisée de 1995 à 1998 portait sur la détection d'erreurs dans des systèmes embarqués critiques. L'originalité de ces travaux porte sur l'utilisation de mécanismes propres au domaine de la sécurité informatique (ici une politique d'intégrité multiniveau), afin de les appliquer à la détection d'erreur dues à des fautes accidentelles. Le but de l'approche est d'assurer la continuité de service du système embarqué et ainsi des propriétés de sécurité-innocuité.

L'étude de ces mécanismes de détection d'erreur dans les systèmes critiques m'a conduit à travailler sur de tels systèmes dans le domaine spatial dans le département R&D d'Astrium à Toulouse. Durant les quatre années où j'ai fait partie de cette entreprise, j'ai mené des activités diverses sur la détection d'erreur ou l'étude de systèmes sûrs de fonctionnement. Ceci m'a par exemple mené au développement de mécanismes de détection d'erreurs sur Colombus (le module européen de la station orbitale internationale), à la définition d'architectures embarquées de satellites autonomes pour le CNES ou à la modélisation de systèmes complexes comme le système d'entrées-sorties d'Ariane 5.

En 2002, j'ai rejoint l'équipe SSIR de Supelec,en effectuant une reconversion de mes thèmes de recherche, puisque j'ai appliqué mes connaissances et mes compétences au domaine de la détection d'intrusion. La découverte des travaux qui avaient été initiés par l'équipe avant mon arrivée m'a permis de m'assurer qu'il y avait un lien fort entre les travaux sur la détection d'erreur en sûreté de fonctionnement et certaines approches

développées en détection d'intrusion. Mes travaux de thèse ayant déjà montré que le lien entre sécurité et sûreté-innocuité étaient pertinents, il m'est naturellement apparu que de tels liens pouvaient sans aucun doute être étendus dans mon nouveau domaine de recherche. Ce constat a donc guidé mes travaux de recherche et s'est déclinée en deux grands types d'études que j'ai conduites ces dernières années : l'utilisation d'une méthode classique de tolérance aux fautes, la diversification fonctionnelle, et l'utilisation de mécanismes classiques de détection d'erreur, les contrôles de vraisemblance. L'objet de mes travaux a été d'étendre ces approches à la détection d'intrusion.

Ce chapitre essaie d'effectuer un bilan factuel le plus exhaustif possible de l'ensemble des activités de recherche que j'ai menées jusqu'à présent, que ce soit en terme d'animation scientifique (section 3.2), de participations à des projets de recherche (section 3.3), d'encadrement de la recherche (section 3.4) ou de publications (section 3.5).

3.2 Animation scientifique

3.2.1 Participations à des comités de programme

Durant les années passées, j'ai participé à quatre comités de programme, dont voici la liste ci-dessous :

- 1st International Conference on Communications and Networking in China (Chinacom 2006), 25-27 octobre 2006, Beijing, China.
- 9th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2007), 14-16 November 2007, Paris, France.
- Computer & Electronics Security Applications Rendez Vous (C&ESAR 2010), 22-24 novembre 2010, Rennes, France .
- 6th Conference on Network Architectures and Information Systems Security (SAR SSI 2011), 18-21 mai 2011, La Rochelle, France.

3.2.2 Implication dans l'évaluation de la recherche (revue par les pairs)

Depuis mon arrivée à Supelec, j'ai participé activement et régulièrement à la rédaction de revue pour des articles de conférences. Certaines de ces conférences comptent parmi les plus réputées du domaine. En voici une liste non exhaustive :

- 19th IFIP International Information Security Conference (IFIP SEC 2004), 23-26 août 2004, Toulouse, France.
- 7th International Symposium on Recent Advances in Intrusion Detection (RAID 2004), 15-17 septembre 2004, Sophia Antipolis, France.
- 11th ACM Conference on Computer and Communications Security (ACM CCS 2004), 25-29 octobre 2004, Washington DC, USA.
- 8th International Symposium on Recent Advances in Intrusion Detection (RAID 2005), 7-9 septembre 2005, Seattle, USA.

- 2006 International Conference on Dependable Systems and Networks (DSN 2006), 25-28 juin 2006, Philadelphia, USA.
- 5th Conference on Security and Network Architectures (SAR 2006), 6-9 juin 2006, Seignosse, France.
- The 5th International Conference on Cryptology and Network Security (CANS 2006), 8-10 décembre 2006, Suzhou, China.
- 2006 IEEE Symposium on Security and Privacy (SSP 2006), 21-24 mai 2006, Oakland, California, USA.
- Eighth International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2006), 17-19 novembre 2006, Dallas, Texas, USA.
- 1st International Conference on New Technologies, Mobility and Security (NTMS 2007), 30 avril-3 mai 2007, Beirut, Lebanon.
- 14th ACM Conference on Computer and Communications Security (ACM CCS 2007), 31 octobre-2 novembre, 2007, Alexandria, VA, USA.
- 15th ACM Conference on Computer and Communications Security (ACM CCS 2008), 27-31 octobre 2008, Alexandria, VA, USA.
- 11th International Symposium On Recent Advances In Intrusion Detection (RAID 2008), 15-17 septembre 2008, Cambridge, Massachusetts, USA.
- 4ème Conférence sur la Sécurité des Architectures Réseaux et des Systèmes d'Information (SAR SSI 2009), 22-26 juin 2009, Luchon, France.
- 2009 International Conference on Dependable Systems and Networks (DSN 2009), 29 juin-2 juillet 2009, Lisbon, Portugal.
- 7th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2010), 8-9 juillet 2010, Bonn, Allemagne.
- 5th Conf. on Network Architectures and Information Systems Security (SAR SSI 2010), 18-21 mai 2010, Menton, France.
- 14th International Symposium on Recent Advances in Intrusion Detection (RAID 2011), 20-21 septembre 2011, Menlo Park, California, USA.
- 7th International ICST Conference on Security and Privacy in Communication Networks (SecureCom 2011), 7-9 septembre 2011, Londres, Angleterre.
- 15th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2012), 12-14 septembre 2012, Amsterdam, Pays Bas.

Par ailleurs, j'ai réalisé deux revues pour des articles de IEEE Transactions on Dependable Computing et IEEE Transactions on Software Engineering.

3.2.3 Participations à des jurys de thèse

Bien que ne possédant pas encore d'habilitation à diriger des recherches, j'ai été amené à participer à deux jurys de thèse, dont l'un en tant que rapporteur, sur dérogation spéciale de l'école doctorale concernée :

- en tant qu'examinateur : thèse de doctorat de l'université de Toulouse de Youssef Laarouchi "Sécurités (immunité et innocuité) des architectures ouvertes à niveaux de criticité multiples : application en avionique", soutenue le 30 novembre 2009 ;

– en tant que rapporteur : thèse de doctorat de Telecom ParisTech de Ludovic Piètre-Cambacédès “Des relations entre sûreté et sécurité”, soutenue le 3 novembre 2010.

3.2.4 Participations à des séminaires

– Journée recherche Supelec Rennes. Détection d'intrusions par corrélation d'événements et d'alertes. Supelec, mars 2004.
– Séminaire DIWALL. Détection d'intrusion par diversification de COTS. Supelec, Octobre 2006.
– Séminaire méthodes formelles et sécurité. Détection d'intrusion au niveau applicatif : approches par invariants. IRISA, Rennes, 24 juin 2011.

3.2.5 Sessions de conférence et animation de club

– 19th IFIP International Information Security Conference (IFIP SEC 2004), Toulouse, France, 23-26 août 2004, session “Authentication”.
– 4th International Conference on Risks and Security of Internet and Systems (CRISIS 2009), Toulouse, France, octobre 2009, session “Attacks, Defense and IDS (II)”.
– Membre du comité de pilotage du club 63 (Système Informatique de Confiance) de la SEE.

3.3 Participations à des projets de recherche

Chacune de mes activités de recherche a été financée au travers de projets de recherche, depuis ma thèse jusqu'à aujourd'hui. Cette section a pour objectif de lister les projets qui ont permis d'obtenir des résultats scientifiques significatifs.

1. 1995-1999 : Projet Européen GUARDS (Generic Upgradable Architecture for Real-time Dependable Systems). Ce projet avait pour objectif de définir une architecture générique de systèmes sûrs de fonctionnement, qui puisse être dérivée aisément dans tous les domaines d'applications critiques. Dans le cadre du projet, l'architecture a été appliquée au ferroviaire, au spatial et au nucléaire. Ma contribution au projet, qui forme mon sujet de thèse, a été de définir une architecture logicielle visant à empêcher la propagation d'erreurs dans les systèmes critiques.
2. 2002-2005 : Projet RNTL DICO (Détection d'Intrusion COopérative). Le but de ce projet était d'étudier des mécanismes de corrélation d'événements et d'alertes dans un système. À Supelec, ce projet a mené à la production d'un prototype de corrélateur d'alertes GNG (Gassata New Generation).
3. 2004-2007 : Projet ACI-SI DADDI (Dependable Anomaly Detection with DIagnosis). Le but de ce projet était de développer des systèmes de détection d'intrusion variés. La contribution de Supelec a été l'étude de mécanismes de détection d'intrusion fondés sur la notion de diversification fonctionnelle.

4. 2006-2008 : Projet RNRT ACES (Analyse Contextuelle d'Evénements de Sécurité). Ce projet avait pour objectif l'étude de mécanismes de corrélation d'alertes dans des systèmes distribués. Cette corrélation pouvait être contextuelle, c'est-à-dire liée à l'état du système distribué. Dans le cadre de ce projet, mon apport a consisté à intégrer le corrélateur GNG à l'intérieur d'un environnement de gestion des alertes *Prelude*.
5. 2007-2010 : Projet ANR POLUX (Policy Unified eXpression). Pour Supelec, l'objectif de ce projet était la configuration automatique de mécanismes de détection d'intrusion se fondant sur la politique de sécurité ou la spécification du logiciel. Au sein de ce projet, nous avons en particulier étudié l'utilisation de mécanismes d'analyse statique offert par l'environnement Frama-C pour la génération automatique au sein du logiciel de contrôles de vraisemblance fondés sur la notion d'invariants.
6. 2008-2012 : Projet ANR DALI (Design and Assessment of application Level Intrusion detection systems). L'objectif de ce projet a été de montrer qu'il était pertinent de penser que des mécanismes de détection d'intrusion spécifiques à chaque application pouvaient fournir des résultats de détection intéressants. Dans ce cadre, nous nous sommes intéressés à la génération automatique de mécanismes de détection d'intrusion au niveau applicatif par apprentissage dynamique de comportements invariants de l'application.

Outre ces projets de recherche à financement publics, j'ai mené durant ces dernières années des projets industriels dont les résultats n'ont pas en soit mené à des résultats scientifiques nouveaux. Toutefois on peut quand même citer :

- Columbus (Agence Spatiale Européenne), spécification, conception et développement du système embarqué du module européen de la station orbitale internationale : conception des mécanismes de détection d'erreurs, de diagnostic et de confinement d'erreurs dans le système embarqué.
- Architecture pour des satellites autonomes (CNES) : définition de l'architecture logicielle de satellites autonomes à haut degré de disponibilité.
- Modélisation en SDL du système d'entrées-sorties d'Ariane 5 (Astrium).
- Environnement virtualisé pour la définition d'une architecture réseau et le test de déploiement d'outils de sécurité (DGA).
- Étude de la sécurité d'une architecture logicielle fondée sur l'environnement CORBA dans le cadre de la radio logicielle (Thales).

3.4 Encadrements

3.4.1 Codirections de thèses

1. Thèse de Cédric Michel, co-encadrée avec Ludovic Mé, sous la direction de Gérardo Rubino,"Langage de description d'attaques pour la détection d'intrusions par corrélation d'événements ou d'alertes en environnement réseau hétérogène", soutenue le 16 décembre 2003 devant le jury composé de Jean-Marc Jezequel (Président),

Frédéric Cuppens (rapporteur), Michel Dupuy (rapporteur), Gérardo Rubino (Examinateur), Ludovic Mé (co-directeur de thèse).

2. Thèse de Frédéric Majorczyk, co-encadrée avec Ludovic Mé,"Détection d'intrusions comportementale par diversification de COTS : application au cas des serveurs web", soutenue le 3 décembre 2008 devant le jury composé de César Viho (Président), Salem Benferhat (Rapporteur), Yves Deswarte (Rapporteur), Eric Totel (co-directeur de thèse), Ludovic Mé (directeur de thèse).
3. Thèse de Jonathan-Christofer Demay, co-encadrée avec Frédéric Tronel, sous la direction de Ludovic Mé,"Génération et évaluation de mécanismes de détection des intrusions au niveau applicatif", soutenue le 1 juillet 2011 devant le jury composé de César Viho (Président), Hervé Debar (Rapporteur), Vincent Nicomette (Rapporteur), Benjamin Monate (Examinateur), Ludovic Mé (directeur de thèse), Eric Totel (co-directeur de thèse), Frédéric Tronel (co-directeur de thèse).
4. Thèse de Olivier Sarrouy, co-encadrée avec Ludovic Mé, "Logiciels auto-testables pour la détection d'intrusion", commencée en 2008, abandonnée en 2009.
5. Thèse de Thomas Demongeot, co-encadrée avec Valérie Viet Triem Tong, sous la direction de Yves Le Traon,"Contrôle de flux d'information dans les architectures orientées services", commencée en 2009, à soutenir en 2012.
6. Thèse de Mounir Assaf, co-encadrée avec Frédéric Tronel et Julien Signoles (CEA), sous la direction de Ludovic Mé,"Vérification de propriétés de sécurité par analyse statique dans des programmes C de grande taille", commencée en 2011, à soutenir en 2014.

3.4.2 Encadrements d'étudiants en stage de master recherche

1. Frédéric Majorczyk, "Détection d'intrusions par diversification fonctionnelle", année 2004.
2. Jonathan-Christofer Demay, "Contrôle de vraisemblance pour la détection d'intrusion", année 2006.
3. Olivier Sarrouy, "Découverte dynamique d'invariants pour la détection d'intrusion", année 2008.
4. Laurent George, "Configuration d'un outil de détection d'intrusion par la politique de sécurité", année 2009.
5. Loïc Le Henaff, "Détection d'attaques contre les données dans les applications web", année 2010.
6. Pierre Karpman, "Amélioration de la génération d'invariants du plugin SIDAN dans Frama-C", année 2012.
7. Erwan Godefroy, "Modélisation du comportement d'une application distribuée pour la détection d'intrusion", année 2012.

3.4.3 Encadrements d'étudiants en stage ingénieur

1. Pierre Martin, stage ingénieur INSA 4ème année, "Développement d'une maquette pour implémenter des tests de sécurité", année 2007.
2. Marco Teamboueon, stage ingénieur INSA 4ème année, "Instrumentation d'un interpréteur BPEL pour le suivi de flux d'information dans les orchestrations de web services", année 2010.

3.5 Publications

Cette section énumère de manière exhaustive l'ensemble de mes publications. Pour une plus grande clarté, les publications les plus importantes sont précédées du symbole ◇.

3.5.1 Mémoire de thèse

Totel E. Politique d'intégrité multiniveau pour la protection en ligne de tâches critiques : Institut National Polytechnique de Toulouse ; 1998.

3.5.2 Revue internationale

Demongeot T, Totel E, Viet Triem Tong V, Le Traon Y. User Data Confidentiality in an Orchestration of Web Services. International Journal of Information Assurance and Security, volume 7.

3.5.3 Chapitre d'ouvrage collectif

◇ Totel E, Beus-Dukic L, Blanquart J-P, Deswarte Y, Nicomette V, Powell D, et al. Multilevel Integrity Mechanisms. In : Generic Upgradable Architecture for Real-time Dependable Systems. Powell D, editor. : Kluwer Academic Publishers ; 2001. p. 99-119.

3.5.4 Conférences internationales avec comité de lecture et actes

Romaric Ludinard, Eric Totel, Frederic Tronel, Vincent Nicomette, Mohamed Kaaniche, Eric Alata, Rim Akrout and Yann Bachy. Detecting Attacks against Data in Web Applications. Proceedings of the 7th International Conference on Risks and Security of Internet and Systems, 2012.

Demongeot T, Totel E, Viet Triem Tong V, Le Traon Y. Preventing data leakage in service orchestration. Proceedings of the International Conference on Information Assurance and Security (IAS 2011), 2011.

◇ Demay J-C, Majorczyk F, Totel E, Tronel F. Detecting illegal system calls using a data-oriented detection model. Proceedings of the 26th IFIP TC-11 International In-

formation Security Conference (IFIP SEC 2011), 2011.

Anceaume E, Bidan C, Gambs S, Hiet G, Hurfin M, Mé L, Piolle G, Prigent N, Totel E, Tronel F, Viet Triem Tong V. From SSIR to CIDre : a New Security Research Group in Rennes. 1st SysSec Workshop, Pays-Bas (2011).

Demay JC, Totel E, Tronel F. SIDAN : a tool dedicated to Software Instrumentation for Detecting Attacks on Non-control-data. Proceedings of the 4th International Conference on Risks and Security of Internet and Systems (CRISIS 2009), 2009.

◇ Sarrouy O, Totel E, Jouga B. Application data consistency checking for anomaly based intrusion detection. Proceedings of the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2009), 2009.

Sarrouy O, Totel E, Jouga B. Building an application data behavior model for intrusion detection. Proceedings of the 23rd Annual IFIP WG 113 Working Conference on Data and Applications Security : Springer (DBSec'2009), 2009.

Demay J-C, Totel E, Tronel F. Automatic Software Instrumentation for the Detection of Non-control-data Attacks. Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection (RAID 2009), 2009.

◇ Majorczyk F, Totel E, Mé L, Saidane A. Anomaly Detection with Diagnosis in Diversified Systems using Information Flow Graphs. Proceedings of the 23rd IFIP International Information Security Conference (IFIP SEC 2008), 2008.

Majorczyk F, Totel E, Mé L. Experiments on COTS Diversity as an Intrusion Detection and Tolerance Mechanism. Proceedings of the First Workshop on Recent Advances on Intrusion-Tolerant Systems (WRAITS 2007), 2007.

Hurfin M, Le Narzul J-P, Majorczyk F, Mé L, Saidane A, Totel E, Tronel F. A Dependable Intrusion Detection Architecture Based on Agreement Services. Proceedings of the Eighth International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2006), 2006.

◇ Totel E, Majorczyk F, Mé L. COTS Diversity based intrusion detection and Application to Web Servers. Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID 2005), 2005.

Totel E, Vivinis B, Mé L. A Language Driven Intrusion Detection System for Event and Alert Correlation. Proceedings of the 19th IFIP International Information Security Conference (IFIP SEC 2004) : Kluwer Academic, 2004.

Totel E, Polle B, Charmeau MC. Modelling an Autonomous Spacecraft Architecture.

Proceedings of the Data Systems In Aerospace (DASIA'01), 2001.

◇ Totel E, Blanquart J-P, Deswarte Y, Powell D. Supporting Multiple Levels of Criticality. Proceedings of the IEEE Symposium on Fault Tolerant Computing Systems (FTCS-28), 1998.

Totel E, Blanquart J-P, Deswarte Y, Powell D. Implementing Safety Critical Systems with Multiple Levels of Integrity. Proceedings of the Data Systems In Aerospace (DASIA'98), 1998.

Totel E, Beus-Dukic L, Blanquart J-P, Deswarte Y, Powell D, Wellings A. Integrity Management in GUARDS. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), 1998.

3.5.5 Conférences nationales avec comité de lecture et actes

Ludinard R, Le Hennaff L, Totel E. RRABIDS, un système de détection d'intrusion pour les applications Ruby on Rails. Dans Actes du Symposium 2011 sur la Sécurité des Technologies de l'Information et des Communications (SSTIC 2011), Rennes, France, 2011.

Demay JC, Totel E, Tronel F. Génération et évaluation de mécanismes de détection d'intrusion au niveau applicatif, Actes de la cinquième conférence sur la Sécurité des Architectures Réseaux et Systèmes d'Information (SARSSI 2010), 2010.

Demongeot T, Totel E, Viet Triem Tong V, Le Traon Y. Protection des données utilisateurs dans une orchestration de Web-Services. Actes de la cinquième conférence sur la Sécurité des Architectures Réseaux et Systèmes d'Information (SARSSI 2010), 2010.

Sarrouy O, Totel E, Jouga B. Un modèle de comportement fondé sur les données pour la détection d'intrusion dans les applications. Actes de la 4ème Conférence sur la Sécurité des Architectures Réseaux et des Systèmes d'Information (SARSSI 2009), 2009.

Majorczyk F, Totel E, Mé L, Saidane A. Détection d'intrusions et diagnostic d'anomalies dans un systéme diversifié par comparaison de graphes de flux d'informations. Proceedings of the 6th Conference on Security and Network Architectures (SARSSI 2007), 2007.

Majorczyk F, Totel E, Mé L. Détection d'intrusions par diversification de COTS. Proceedings of the 4th Conference on Security and Network Architectures (SAR 2005), 2005.

Totel E. Coexistence de logiciels de différents niveaux de criticité dans des systèmes distribués. Actes des Journées Doctorales en Informatique et Réseaux (JDIR 98), 1998.

3.5.6 Rapports techniques

Totel E, Majorczyck F, Mé L. COTS Diversity based Intrusion Detection. Supelec, 2005.

Vivinis B, Totel E, Mé L. Sémantique du langage de description d'attaques ADeLe. Supelec, 2005.

Blanquart JP, Deswarte Y, Powell D, Totel E. Multilevel Integrity Mechanisms. Rapport GUARDS D1A3/AO/2009-D, rapport LAAS n° 97034, 1997.

Totel E, Deswarte Y, Powell D. Suitability of Operating Systems Considered in GUARDS. Rapport GUARDS I1SA1/TN/5006/A, 1997.

Totel E. Experimental Implementation of Integrity Mechanisms on CORBA. Rapport GUARDS I1SA1/TN/5007/A, 1997.

3.5.7 Rapports de contrats

Totel E, Tronel F. Livrable DALI 2.2 : Mécanismes de détection d'intrusion fondés sur les invariants. Rapport ANR DALI, mai 2011.

Totel E et al. Livrable POLUX 3.1 : Maquette et configuration des services de sécurité. Rapport ANR POLUX, janvier 2011.

Totel E, Vivinis B. Étude des impacts de l'utilisation de CORBA sur la sécurité. Rapport de contrat pour Thalès, 2008.

Majorczyk F, Totel E, Mé L. Livrable DADDi 3.1 : Approches implicites. Rapport DADDi, 2007.

Morin B. et al. Livrable ACES 6.1 : Spécifications détaillées du gestionnaire de corrélation. Rapport ACES, 2007.

Vivinis B, Totel E, Mé L. Livrable DICO : Langage de description d'attaques ADeLe. Rapport DICO, avril 2005.

Goubault-Larrecq J, Ducassé M, Pouzol JP, Demri S, Olivain J, Picaronny C, Mé L, Totel E, Vivinis B. Livrable DICO SP 3.3 : Algorithmes de détection et langages de signatures. Rapport DICO, 8 octobre 2003.

Deuxième partie

Contributions scientifiques

Chapitre 1

Introduction

L'ensemble des travaux de recherche que j'ai menés ces dix dernières années porte sur la détection d'intrusion. J'ai initié ces travaux en 2002 lors de mon arrivée dans l'équipe SSIR (Sécurité des Systèmes d'Information et Réseaux). À cette époque, j'ai participé au projet DICO (Détection d'Intrusion COopérative) au sein duquel il m'a été donné de développer des mécanismes de corrélation d'alertes fondés sur la définition d'une base de signatures d'attaques dans un langage créé pour l'occasion : ADeLe (Attack Description Language). Ce langage est le résultat de la thèse de Cédric Michel que j'ai co-encadrée avec Ludovic Mé. Ces travaux ont mené au développement d'un outil de corrélation d'alertes : *GNG*.

Cette introduction au domaine de la détection d'intrusion m'a permis de constater que la notion de mécanisme de détection d'intrusion telle qu'on la perçoit en sécurité est très proche de la notion de mécanisme de détection d'erreur telle qu'on la définit dans le domaine plus large de la sûreté de fonctionnement. Ce constat m'a amené à démarrer trois thèses ayant pour but de montrer l'efficacité de mécanismes de détection d'erreur bien connus en sûreté de fonctionnement dans le domaine de la détection d'intrusion. La première thèse, réalisée par Frédéric Majorczyk, a démarré en septembre 2004. Elle a pour sujet la détection d'intrusion fondée sur des solutions architecturales logicielles à base de diversification fonctionnelle. Les deux autres thèses, menées par Jonathan Christopher Demay et Olivier Sarrouy, ont démarré en janvier 2006 et septembre 2008, la deuxième ayant avorté en 2009. Ces deux thèses portaient sur l'utilisation de contrôles de vraisemblance pour la détection d'intrusion dans le logiciel, chacune de ces thèses appliquant une méthode différente pour la génération automatique des mécanismes de détection.

Dans ce mémoire, j'ai choisi pour une plus grande cohérence de présenter une sélection de mes travaux. Le choix s'est porté sur les travaux qui, à mon sens, ont présenté une certaine originalité par rapport aux travaux classiques menés en détection d'intrusion. Ces travaux sont ceux qui reposent sur des mécanismes de détection d'erreur au sens général du terme. Dans la suite de ce document, je vais montrer les relations

entre détection d'intrusion et détection d'erreur et faire un bref état de l'art du domaine de la détection d'intrusion au niveau applicatif (cf. chapitre 2). Ensuite, je détaillerai les deux grands types de travaux que j'ai menés en détection d'intrusion : l'utilisation de la diversification fonctionnelle d'une part (cf. chapitres 3 et 4) et l'utilisation de contrôles de vraisemblance d'autre part (cf. chapitres 5 et 6).

Chapitre 2

Détection d'intrusion et détection d'erreur

Comme mentionné précédemment, le but des travaux présentés dans ce document est de montrer que des techniques utilisées dans le domaine de la sûreté de fonctionnement pour détecter des erreurs peuvent être utilisées dans le domaine de la sécurité pour détecter des intrusions. Pour montrer la pertinence de cette approche, il est nécessaire de faire un parallèle entre les concepts définis en sûreté de fonctionnement et ceux définis en sécurité. C'est le sujet de la section 2.1. Nous montrons ensuite (section 2.2) que certaines approches utilisées en détection d'intrusion tirent déjà profit de ce parallèle, sans pour autant le justifier. Ceci est tout particulièrement vrai lorsqu'on s'intéresse à la détection d'intrusion au niveau applicatif, c'est-à-dire lorsque l'on cherche à détecter des intrusions qui visent à corrompre le fonctionnement d'une application en particulier.

2.1 Concepts

Dans le domaine de la sûreté de fonctionnement, la terminologie définie par [ALRL04] présente de manière claire les entraves qui peuvent mener à un mauvais fonctionnement d'un système. Ainsi, trois types d'entraves causalement dépendantes sont définies ; les *fautes* d'une part, qui sont la cause initiale d'un mauvais fonctionnement ; les *erreurs* qui sont les conséquences directes de l'activation des fautes et qui peuvent produire causalement d'autres erreurs par propagation dans le système ; ces erreurs peuvent enfin induire des *défaillances* qui caractérisent la sortie du système de sa spécification. Ce qui est particulièrement intéressant dans cette terminologie est qu'elle s'applique quelles que soient les types de fautes à l'origine des erreurs puis défaillances. Ainsi, ces fautes peuvent être internes au système (des *bugs* résiduels, ou fautes de conception) ou externes. Et parmi ces fautes externes, certaines peuvent être accidentelles ou intentionnelles. C'est ce dernier cas qui nous intéresse en sécurité. En effet, dans ce domaine, les fautes qui influencent l'état du système sont souvent des fautes externes intentionnelles avec volonté de nuire au système dans le but de violer ses propriétés de sécurité. Ces fautes sont alors appelées des attaques.

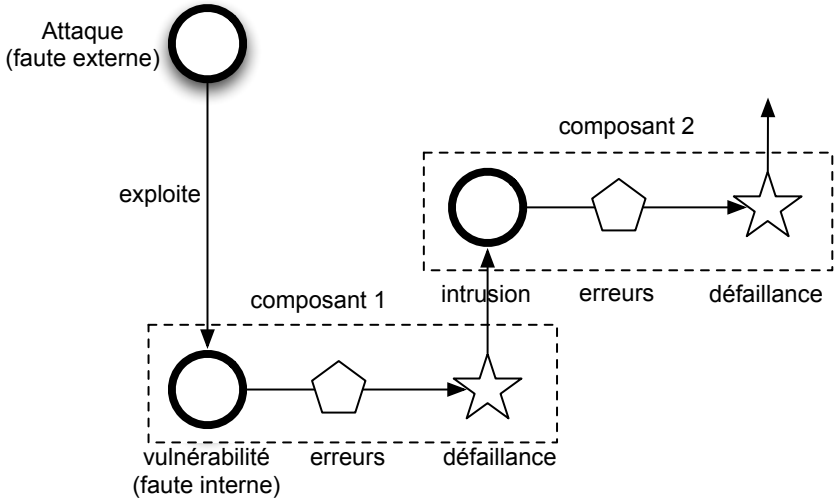


FIGURE 2.1 – Notion d'Attaque-Vulnérabilité-Intrusion

La dérivation de cette terminologie a été clairement exprimée dans le domaine de la sécurité dans le cadre du projet MAFTIA [AAC+03]. Lorsqu'on parle d'entraves au bon fonctionnement du système en sécurité, on considère le triptyque attaque, vulnérabilité, intrusion (cf. figure 2.1). Une attaque, telle qu'elle a été définie dans le paragraphe précédent est la cause première de la réussite éventuelle d'intrusion dans le système. Dans MAFTIA, l'intrusion se définit comme une faute résultante d'une attaque réussie contre le système. Afin de réussir cette attaque, il est nécessaire d'exploiter une faute de conception du système qu'on nomme une vulnérabilité. Ainsi, l'intrusion se caractérise comme le résultat d'une attaque qui exploite avec succès une vulnérabilité. C'est donc la composition de l'exploitation de fautes qui implique un potentiel mauvais fonctionnement du système.

Cette terminologie explique le lien entre les différentes entraves aux propriétés de sécurité du système. Toutefois elle se différencie de ce qui est généralement admis en sécurité, où on définit généralement l'intrusion comme une violation de la politique de sécurité du système. En un sens, son pouvoir de modélisation est supérieur, car une attaque réussie peut ne pas mener de manière immédiate à une violation de la politique. Par exemple, un attaquant qui réussit à s'authentifier en utilisant un compte et un mot de passe valides ne viole pas en soit la politique de sécurité du système, mais le fera lorsqu'il effectuera une élévation des privilèges obtenus afin de lire des informations sensibles.

En projetant le triptyque attaque/vulnérabilité/intrusion sur la notion de faute telle

qu'elle est définie dans le domaine de la sûreté de fonctionnement, le modèle considère que l'activation de fautes entraîne la production d'erreurs dans le système. Face à ces erreurs, l'un des moyens d'assurer le bon fonctionnement d'un système est de mettre en place des mécanismes de tolérance aux fautes. Ces mécanismes ont pour objectif d'assurer le bon fonctionnement, même en présence d'erreurs, et incluent souvent des mécanismes de détection d'erreurs. Ceux-ci ont pour rôle de détecter des états incorrects du système, soit pendant son fonctionnement, soit après une défaillance dans le cadre d'une analyse post-mortem. La notion de mécanismes de détection d'erreur peut donc tout naturellement être étendue à des erreurs dues à des attaques. On nomme ce type de détection la détection d'intrusion.

Alors que le domaine de la détection d'intrusion se définit usuellement comme l'ensemble des techniques visant à détecter des violations de la politique de sécurité, la définition que propose le projet MAFTIA est sensiblement différente. Il s'agit en fait de l'ensemble des techniques et mécanismes permettant de détecter des erreurs qui peuvent conduire à la violation de la politique de sécurité, ou permettant de diagnostiquer des attaques. Cette définition, bien que se focalisant également sur les conséquences des attaques sur le système, introduit la notion d'erreurs intermédiaires avant violation de la politique de sécurité. Et ces erreurs peuvent être sensiblement les mêmes, que leurs causes soient des fautes accidentelles ou intentionnelles, c'est-à-dire qu'elles apparaissent dans un contexte d'attaque ou dans un contexte accidentellement erroné. Nous retrouverons cette similarité dans l'ensemble des travaux que nous présentons dans ce mémoire.

2.2 La détection d'intrusion

La détection d'intrusion est un domaine qui est apparu en 1980 grâce au rapport de [And80]. De nombreux travaux ont été menés par la suite sans qu'il n'y ait réellement d'uniformisation des termes utilisés dans le domaine. Toutefois, [DDW99] propose une taxonomie des systèmes de détection d'intrusion (IDS) sur laquelle je vais me fonder dans la suite de cette section.

2.2.1 Caractéristiques des IDS

Un IDS se caractérise par un certain nombre de propriétés :

1. la méthode de détection qu'il met en œuvre ;
2. le comportement lors de la détection : l'IDS peut simplement lever des alertes (IDS passif) ou réagir aux attaques ou intrusions détectées (IDS actif ou IPS) ;
3. la source de données dont se sert l'IDS : cela peut être des logs systèmes ou applicatifs, des fichiers systèmes (on parle alors d'*IDS hôtes*), des paquets réseaux (on parle alors d'*IDS réseau*). Suivant la source de données utilisée, l'IDS pourra bien entendu être plus efficace pour détecter certains types d'attaques plutôt que d'autres ;

4. la fréquence d'analyse : l'IDS peut effectuer sa détection en temps réel ou de manière périodique.

Les travaux que je présente dans ce mémoire utilisent tous des sources de données systèmes ou applicatives. Ce sont donc des IDS hôtes. D'autre part, leur comportement consiste uniquement à lever des alertes sans appliquer de mécanismes de réaction automatique.

2.2.2 Méthodes de détection

Lorsqu'on parle de système de détection d'intrusion, on pense avant tout à la méthode de détection utilisée. Dans la pratique, on distingue deux grands types de méthodes : l'approche à base de connaissances ou approche par signatures (*misuse detection*) et l'approche comportementale (*anomaly detection*).

L'approche par signatures nécessite d'avoir la connaissance des attaques auxquelles le système est exposé. Cette base de connaissance est utilisée pour détecter l'occurence d'attaques contre le système. Ce type d'approche reste encore aujourd'hui très utilisé dans les systèmes commerciaux, alors qu'elle présente d'importants inconvénients. En particulier, cette méthode ne peut bien évidemment pas détecter d'attaques inconnues ou non répertoriées, ce qui signifie que la base de signatures doit constamment être mise à jour. Cette opération peut être considérée comme un défi compte tenu du nombre de nouvelles attaques qui apparaissent régulièrement.

L'approche comportementale consiste au contraire à modéliser le comportement normal du système ou de la partie du système qui risque d'être attaquée. Ce comportement est utilisé dans la phase de détection en observant si le système dévie de ce comportement normal. Si tel est le cas, on suppose qu'une attaque a eu lieu contre le système. Le principal avantage de cette approche par rapport à l'approche par signatures est qu'elle est capable de détecter des attaques non encore connues. Toutefois, la difficulté de l'approche comportementale consiste en la construction d'un modèle de comportement légitime exact et complet. Sans ces deux propriétés, la détection risque d'être sujette à des faux positifs (des détections alors qu'il n'y a pas eu d'attaques) ou à des faux négatifs (des attaques non détectées par le système de détection d'intrusion).

Les travaux que j'ai menés portent tous sur des approches comportementales, méthode qui me semble aujourd'hui la plus prometteuse parce que ne nécessitant pas de mise à jour constante de l'IDS ou de sa configuration.

2.2.3 Approche comportementale : détection par la spécification

La mise en place d'une méthode de détection comportementale nécessite de modéliser le comportement normal du système ou d'une partie du système. Le problème qui se pose alors est de déterminer comment construire ce modèle de comportement. Comme ici nous nous intéressons à la détection d'intrusion, on pourrait s'attendre à ce que le

comportement soit en fait une spécification de règles de sécurité qui doivent être imposées au système. C'est par exemple le cas de [HTMM09, ZMB03] où le modèle de comportement est en fait une politique de flux d'information, c'est-à-dire une spécification des flux d'information autorisés au sein du système. L'IDS détecte toute violation de cette politique et lève une alerte dans ce cas. De même, dans [KFL94], les auteurs spécifient un comportement pour chaque application sensible du système. Ce comportement consiste en l'interaction des programmes avec le reste du système (ouvertures de fichiers, exécution de programmes). Il représente les actions autorisées d'une application en interaction avec son environnement. Une violation de ce comportement indique une potentielle attaque. Dans [KRL97], les auteurs spécifient des séquences autorisées d'actions dans des traces produites dans des systèmes distribués et vérifient la validité de ces traces pour détecter des intrusions. Cette approche limite la spécification des séquences à des comportements en lien avec la sécurité du système d'information.

Toutefois, l'utilisation d'une spécification "fonctionnelle" (c'est-à-dire ne se limitant pas à des règles de sécurité) comme modèle de comportement a également été utilisée en détection d'intrusion. De telles approches se rapprochent fortement du domaine du domaine du test où il est usuel d'utiliser un modèle de la spécification durant la phase de test, afin de détecter des erreurs ou défaillances du système. En détection d'intrusion, ce type d'approche a été particulièrement utilisé au niveau réseau, pour détecter des attaques contre des protocoles comme OLSR [TSB+05] ou AODV [TBK+03].

Il est important de noter ici que toute spécification, complète ou non, d'un système ou d'un logiciel est susceptible d'être utilisable en détection d'intrusion pour peu que des attaques puissent conduire à sa violation de par leur action. Cette spécification peut exprimer les fonctions normales que doit réaliser le logiciel, tout autant que des propriétés non fonctionnelles (de sécurité par exemple). Lorsqu'en particulier on parle de détection d'intrusion au niveau applicatif, on modélise souvent le comportement fonctionnel de l'application sans introduire des comportements "de sécurité".

2.3 La détection d'intrusion comportementale au niveau applicatif

Dans nos travaux, nous nous intéressons tout spécialement aux conséquences d'une attaque contre une application. La détection comportementale au niveau applicatif passe par la construction d'un modèle de référence de l'application qu'on souhaite protéger. Suivant ce qu'on est capable d'observer, le comportement de référence porte sur des aspects différents de l'exécution du programme. Ainsi, si on observe l'interaction du programme depuis le système d'exploitation (approche en boîte noire), le modèle portera sur les appels système. Si on a accès à l'intérieur de l'application (approche en boîte blanche), le modèle pourra porter sur le chemin d'exécution parcouru au sein de l'application, ou les contenus des variables de l'application. Dans la suite de ce chapitre, nous passons en revue les différentes méthodes permettant de détecter des

attaques contre une application.

2.3.1 Détection d'intrusion au niveau système

Observation du flot de contrôle L'approche la plus classique au niveau système est celle proposée par [HFS98]. Cette méthode consiste à vérifier que l'enchaînement des appels systèmes émis par une application est conforme à celui préalablement appris durant une phase d'apprentissage réalisée dans un environnement exempt d'attaque. Toute la difficulté de l'approche consiste à se demander quelle est la taille de la séquence d'appels qui doit être retenue pour réaliser la détection. Cette approche propose donc de construire par apprentissage un comportement fonctionnel de l'application (l'ordre de l'enchaînement des appels systèmes) dont on imagine qu'il sera violé lors d'une attaque contre l'application. Dans la pratique, ce genre d'approche est sujet à des méthodes de contournement (dites par mimétisme), où l'attaquant s'arrange pour que le comportement de l'application attaquée soit le même que son comportement normal. Par exemple, du code injecté exécuté par l'application réalise volontairement la même séquence d'appels système que l'application durant une exécution normale. Un autre exemple serait l'exécution d'un appel système correct, mais avec des données invalides.

Afin de pallier ce défaut, un certain nombre d'approches ont vu le jour. [GRS04b] propose une approche en *boîte grise* qui consiste à ajouter un contexte à chaque appel système afin de pouvoir le discriminer de manière plus précise. Une première proposition consiste à s'intéresser au compteur ordinal admissible pour un appel système donné, ce qui permet de s'assurer que l'appel système est invoqué par le code du programme et non par du code injecté. Une deuxième proposition consiste à surveiller la pile d'appels du processus et plus particulièrement les différentes adresses de retour présentes sur celle-ci. Chaque appel système est donc associé à un état de la pile, qui est vérifié à l'exécution pour s'assurer de sa validité.

L'approche développée dans [MRVK07] consiste à vérifier la validité des arguments des appels système émis par le processus surveillé. Cette technique a pour objectif de complexifier les attaques par mimétisme qui visent à modifier les données manipulées par le processus. Elle consiste en fait à associer une valeur utilisée par un appel système à un contexte de pile, comme dans l'approche précédente. Si pour un contexte donné l'ensemble des valeurs que peut prendre un argument est très petit, la capacité de détection de la compromission de l'argument est très forte.

Dans [GRS04a], les auteurs décrivent une façon d'extraire par apprentissage un graphe de contrôle d'une application et de définir pour chaque appel système dans quel état de ce graphe il est légal. Cette approche ne nécessite pas d'avoir à disposition le code de l'application et repose sur l'existence de la pile d'appels. À chaque appel effectué, l'IDS vérifie si l'appel système est effectué dans un contexte légal du point de vue du graphe de contrôle préalablement construit.

Observation du flot de données Les approches précédentes se focalisent uniquement sur le flot de contrôle d'une application vu depuis le système. L'inconvénient de telles approches est qu'elles sont vulnérables à des attaques contre des données qui ne modifient pas le flot de contrôle [CXS+05]. Afin de pallier cet inconvénient, [BCS06] propose de définir la notion de flot de données entre les arguments des appels systèmes. Cette notion est exprimée par des règles reliant les différents appels systèmes du programme à l'exécution. Les auteurs montrent sur un certain nombre d'attaques l'efficacité de leur approche.

Ce qui ressort de ces approches est plutôt intéressant : en aucun cas on ne parle ici de règles de sécurité à vérifier. Ces approches ont toutes le même objectif : construire un modèle fonctionnel de l'application qui soit le plus précis possible (sans inclure la moindre notion non fonctionnelle telles que des règles de sécurité), afin d'être capable d'en détecter toute déviation à l'exécution. Ce que l'on détecte n'est donc pas forcément des intrusions, mais bel et bien des erreurs qui pourraient aussi bien être dues à des fautes accidentelles qu'à des attaques.

2.3.2 Détection d'intrusion dans l'application

Grâce à des mécanismes de prévention tels que les pare-feux, les systèmes d'information ne sont accessibles à distance qu'au travers de logiciels serveurs (pour lesquels certains ports spécifiques sont ouverts), qui deviennent ainsi la cible des attaques. Dans la section précédente, nous considérions le logiciel comme une *boîte noire*, et n'envisagions donc que ses interactions avec le système. Lorsqu'on considère le logiciel comme une *boîte blanche*, on a cette fois accès à son fonctionnement interne. Il peut donc être judicieux d'essayer de détecter des conséquences d'attaques directement au sein du logiciel afin d'éviter toute éventuelle propagation d'erreurs vers le système ou d'autres logiciels en interaction avec le programme attaqué.

Que les fautes soient accidentelles ou intentionnelles, les erreurs découlant de ces fautes au sein du logiciel peuvent être catégorisées en deux classes distinctes. La première classe concerne l'intégrité du flot de contrôle d'une application. La seconde classe concerne l'intégrité des données manipulées par une application. Dans la suite de cette section, nous abordons la détection de ces deux types d'erreurs.

Détection de la violation du flot de contrôle La détection de la violation du flot de contrôle dans une application a pour objectif de vérifier que le code de l'application s'exécute dans le bon ordre ou que le code exécuté est bien le code de l'application considérée. Dans le premier cas, on vérifie que l'on n'effectue pas des branchements incorrects au sein de l'application. Dans le deuxième cas, on s'assure que le flot de contrôle n'est pas transféré en dehors du code originel de l'application.

Les travaux sur la vérification de l'intégrité du flot de contrôle dans les applications sont assez nombreux. On peut citer [GRRV03, VA06] qui sont de nature proche et qui

ciblent essentiellement la détection d'erreurs dues à des fautes accidentelles. Ces deux approches divisent le code en blocs et des vérifications du flot de contrôle sont insérées en début de chaque bloc. Ces vérifications reposent sur une notion de signature. Ces signatures sont calculées de manière statique aux différents points du programme où leur vérifications doivent être effectuées. À l'exécution, on calcule la valeur que devrait prendre la signature compte-tenu du chemin parcouru dans le programme, et on vérifie que cette valeur est conforme à celle calculée avant l'exécution. Toute la difficulté de l'approche consiste en la rapidité de calcul de la signature, afin de limiter l'impact sur les performances du programme exécuté.

Les travaux menés dans [ABEL03] et [ABEL05] s'intéressent à la détection de la violation du flot de contrôle dans un contexte de fautes intentionnelles, en supposant que l'attaquant peut changer arbitrairement des données en mémoire. Cette méthode repose sur un encodage du graphe de contrôle du programme en embarquant des vérifications dans le logiciel, comme dans les méthodes précédentes. Toutefois dans cette méthode, les vérifications sont évaluées à la destination des branchements, et non pas à leur source. Il faut noter que l'insertion des vérifications s'effectue dans le binaire directement et non pas dans le code source comme pour la plupart des méthodes. Cela permet donc d'instrumenter un programme dont on ne possède pas le code source. Afin de prendre en compte les attaques, cette approche repose sur trois hypothèses : l'unicité des identificateurs sur lesquels repose la vérification du graphe de contrôle ; l'impossibilité d'écrire dans une zone de code ; l'impossibilité d'exécuter du code dans une zone de données. Ces hypothèses semblent raisonnables mais peuvent devenir un problème dans le cas de code auto-modifiable, de la génération automatique de code à l'exécution, ou du lien dynamique avec du code chargé à l'exécution. Si ces hypothèses sont vérifiées, les contrôles de peuvent pas être évités, et on traite correctement le problème des attaques.

Toutes les approches préalablement proposées reposent sur l'instrumentation du code source ou du code binaire afin d'insérer les mécanismes de vérification de l'intégrité du flot de contrôle de l'application. Toutefois on peut envisager que les mécanismes de vérification soient présents dans un moniteur externe à l'application. C'est le cas des travaux décrits dans [KBA02]. Contrairement aux papiers précédents, les auteurs ne reposent pas sur des hypothèses permettant d'assurer des propriétés de sécurité au programme exécuté. Par exemple, ils considèrent que du code injecté peut être exécuté sur la machine, grâce à des vulnérabilités classiques. Le but de leur approche consiste à observer à l'exécution tous les transferts dans le flot de contrôle, et à vérifier que ces transferts satisfont une politique de sécurité. Ceci est réalisé en utilisant trois techniques : la restriction des codes d'origine, la restriction des transferts du contrôle, et un bac à sable incontournable (*sandbox* en anglais). La politique de sécurité concerne un sous-ensemble spécifié du graphe de contrôle de l'application. Un moniteur dans un interpréteur est responsable de la vérification de la politique de sécurité. L'implémentation se fait en fait dans un RIO (Runtime Introspection and Optimization) d'un processeur IA-32.

Détection de la violation du flot de données Les approches par contrôle de flot d'exécution permettent de détecter des erreurs qui se caractérisent par des modifications du graphe de flot de contrôle du programme. Mais elles se limitent malheureusement à ce modèle de fautes. Certaines attaques, lorsqu'elles sont réalisées, peuvent pourtant n'avoir aucun impact sur le flot de contrôle du programme tout en permettant d'atteindre des objectifs similaires. De telles attaques existent et ont été mises en évidence par [CXS+05].

Afin de détecter des attaques de ce type sur les données, des solutions fondées sur l'analyse du flot de données dans un programme ont été développées. Par exemple, les travaux conduits dans [CCH06] définissent un graphe de flot de données préalablement à l'exécution du programme. Ce graphe décrit pour chaque donnée l'ensemble des instructions qui pourraient écrire dans cette donnée. À chaque lecture de la donnée, on vérifie que les instructions ayant écrit dans cette donnée sont bien celles définies par le graphe initial. Ceci permet de détecter des écritures non désirées dans des variables grâce à l'exploitation de vulnérabilités telles qu'un *buffer overflow*, une vulnérabilité de type *format string*, etc. Le surcoût induit par les vérifications à l'exécution a été mesuré en utilisant le framework SPEC. Les résultats montrent que l'approche peut être assez coûteuse en temps processeur. [ACR+08] a un objectif similaire et met en place des mécanismes de vérification du flot de données comparables.

2.3.3 Bilan des méthodes présentées

	Attaques contre le flot de contrôle	Attaques contre les données
OS	[HFS98], [GRS04b], [MRVK07], [GRS04a]	[BCS06]
Application	[GRRV03], [VA06], [ABEL05], [KBA02]	[CCH06], [ACR+08]

Table 2.1 – Détection d'intrusion au niveau applicatif

Dans la table 2.1, on peut voir les attaques détectées par les méthodes que nous avons préalablement présentées. On remarquera en particulier que peu de méthodes s'intéressent aux attaques contre les données, que ce soit au niveau système ou au niveau de l'application. Il faut toutefois noter que les pertes de performances qu'elles induisent au niveau application à l'exécution sont un frein certain à leur utilisation.

2.4 Conclusion

Les travaux décrits dans ce mémoire tournent tous autour de la détection d'attaques contre des applications. La vision choisie varie suivant les approches proposées. Ainsi les chapitres 3 et 4 considèrent l'application comme une boite noire et s'intéressent aux interactions de l'application soit avec un client sur le réseau, soit avec le système d'exploitation qui l'héberge. Les chapitres 5 et 6 adoptent une vision interne au logiciel et s'intéressent à la violation éventuelle d'invariants au sein de l'application à l'exécution.

Par contre, les différents travaux adoptent des mécanismes de détection d'erreur issus du domaine de la sûreté de fonctionnement, qui reposent sur la détection de violation des fonctionnalités de l'application et non sur la violation de règles de sécurité. On verra que pourtant, cette vision générale est pertinente dans le cadre de la détection d'intrusion.

Chapitre 3

Détection d'intrusion par diversification fonctionnelle

3.1 Introduction

Comme il a été présenté dans l'état de l'art, nous nous focalisons dans ces travaux sur une détection comportementale des intrusions. Plus précisément, nous nous intéressons à un modèle du comportement qui est une spécification fonctionnelle de l'application. Dans la pratique, il est difficile de construire une spécification suffisamment précise et complète qui puisse servir de modèle de référence à l'exécution de l'application. Compte-tenu de la complexité d'un logiciel, une telle approche risque fatalement d'entrainer un grand nombre de faux négatifs et de faux positifs.

Afin de contrer un tel problème, il faudrait en fait une spécification qui soit aussi complète dans sa description que le logiciel lui-même. C'est le fondement de la notion de diversification fonctionnelle, et en particulier de la programmation N-Version : il s'agit de disposer de plusieurs logiciels écrits à partir des mêmes spécifications fonctionnelles. Chacun de ces logiciels peut servir de modèle de référence pour vérifier à l'exécution le bon fonctionnement de l'application considérée.

Dans ce chapitre, nous présentons notre approche (section 3.2) et l'implémentation que nous en avons faite pour la détection d'intrusion dans le cadre de serveurs web (section 3.3) ainsi que les résultats que nous avons obtenus. Enfin, nous nous positionnerons par rapport à des travaux existants (section 3.4).

Les travaux présentés dans ce chapitre ont été publiés dans RAID 2005 [TMM05].

3.2 Détection d'intrusion par diversification fonctionnelle

La diversification fonctionnelle est une méthode issue de la sûreté de fonctionnement qui a pour objectif de détecter et tolérer des fautes. Dans cette section, nous

présentons en détail la diversification fonctionnelle, et en particulier la programmation N-Versions. Ensuite, nous nous focalisons sur les problèmes qui dérivent de l'utilisation de composants sur l'étagère (COTS pour *Components off the Shelf*) dans une architecture N-Versions. Enfin, nous détaillons l'architecture que nous proposons.

3.2.1 Principes de la diversification fonctionnelle

La diversification fonctionnelle est une technique applicable à tous les éléments d'un système d'information : le matériel, le logiciel, les liens de communication, etc. Lors de la mise en place d'une simple réplication de composants dans le système afin d'assurer des propriétés de tolérance aux fautes, on s'expose à des fautes de mode commun, c'est-à-dire au risque que les mêmes fautes soient activées dans les différents composants répliqués lorsqu'ils sont plongés dans le même environnement et exposés aux mêmes sollicitations. Afin de palier ce problème, la diversification fonctionnelle consiste à proposer des composants offrant les mêmes fonctionnalités (c'est-à-dire ayant exactement la même spécification), mais conçus et réalisés de manière complètement indépendante. Cette indépendance est sensée assurer la décorrélation des fautes de conception entre les répliques, afin de réduire la probabilité de fautes de mode commun.

Il existe trois types de diversification fonctionnelle : les blocs de recouvrement [Ran75], la programmation N-Autotestable [LABK90] et la programmation N-Versions [AC77]. Dans notre cas, c'est sur la programmation N-Versions que nous nous focalisons, les différentes versions étant en fait des composants sur l'étagère développés indépendamment les uns des autres.

La programmation N-Versions consiste à développer des logiciels indépendants du point de vue de leurs fautes de conceptions résiduelles, tout en ayant la même spécification. Toutefois, obtenir une telle indépendance n'est pas chose aisée, car elle nécessite une conception et une implémentation du logiciel qui soient complètement différentes (équipes de programmeurs différents, outils de conception différents, langages de programmation différents, etc.). L'utilisation de la programmation N-Versions se fait principalement dans des contextes où l'on veut assurer la tolérance aux fautes et la continuité de service du système, ce qui est réalisé en fournissant aux différentes variantes les mêmes entrées, et en traitant les sorties. Cette approche permet donc de détecter ou tolérer les fautes qui ont une influence sur les sorties. Le traitement sur les sorties consiste dans la plupart des cas en un mécanisme de vote qui est capable de décider quelles sorties des différentes variantes doivent être utilisées par le reste du système. Ainsi, même en présence d'un certain nombres de variantes défaillantes, le système continue à fournir une sortie correcte. Dans la pratique, et grâce à l'hypothèse de la décorrélation des fautes entre les variantes, si f variantes sont défaillantes, f+2 variantes sont nécessaires pour assurer la continuité de service du logiciel diversifié. Un comparateur sur les sorties détermine quelle est la réponse correcte (c'est-à-dire celle existant en au moins deux exemplaires).

Des études expérimentales ont montré que l'utilisation de la programmation N-Versions est extrêmement efficace, qu'elle soit utilisée comme moyen de tolérance (au moins trois variantes sont disponibles sous l'hypothèse qu'une seule d'entre elles peut défaillir à la fois), ou comme simple moyen de détection d'erreurs (au moins deux variantes sont disponibles, une seule pouvant être amenée à défaillir à un instant donné).

3.2.2 Diversification de COTS

Bien qu'extrêmement efficace, la programmation N-Versions est extrêmement coûteuse car elle nécessite le développement de N variantes d'un même logiciel, par N équipes différentes. Alors que le coût peut sembler dérisoire lorsqu'il s'agit de systèmes critiques mettant en jeu des vies humaines, il est plus difficile de le faire accepter dans le cadre de la sécurité informatique. Cependant la plupart des services accessibles sur internet (serveurs web, ftp, etc.) existent naturellement sur différents types d'environnements (différents systèmes opératoires en particulier), voire existent sous la forme de logiciels propriétaires ou d'implémentations libres. On a donc à notre disposition une diversification naturelle des services disponibles, et souvent à très faible coût.

L'idée qui consiste à utiliser ces composants sur l'étagère, pour construire un système fondé sur la diversification, semble donc tout à fait naturelle. Toutefois, selon les hypothèses liées à la programmation N-Versions, toutes les variantes doivent avoir la même spécification. Or dans le cas de COTS, rien ne permet de prétendre que tous les services utilisés dans l'architecture ont bien la même spécification. En fait, la seule spécification dont on est sûr qu'elle est partagée par toutes les variantes est celle qui décrit l'interaction du logiciel avec son client, c'est-à-dire ses interfaces. En général les services ont des interfaces définies par des standards (en particulier lorsqu'il s'agit de protocoles d'interaction).

D'autre part, la diversification fonctionnelle a pour objectif de produire des variantes dont les fautes de conception résiduelles sont indépendantes. Malheureusement l'utilisation de COTS ne permet pas de s'assurer facilement qu'une telle propriété est vérifiée, sauf à tester dynamiquement et empiriquement que les vulnérabilités connues dans chacune des variantes ne sont pas présentes dans les autres. Cette approche serait loin d'être exhaustive, compte-tenu du fait qu'il peut également potentiellement exister des vulnérabilités non encore connues.

Afin de pouvoir mettre en place un mécanisme de vote ou une comparaison sur les sorties, il faut également que ces sorties soient décrites par la spécification commune des différents COTS. C'est donc cette spécification qui va dicter quelles sont les sorties qui doivent être comparées, comment elles doivent être comparées et à quel moment elles doivent être comparées. Ces choix auront un impact fort sur la capacité de détection du "comparateur".

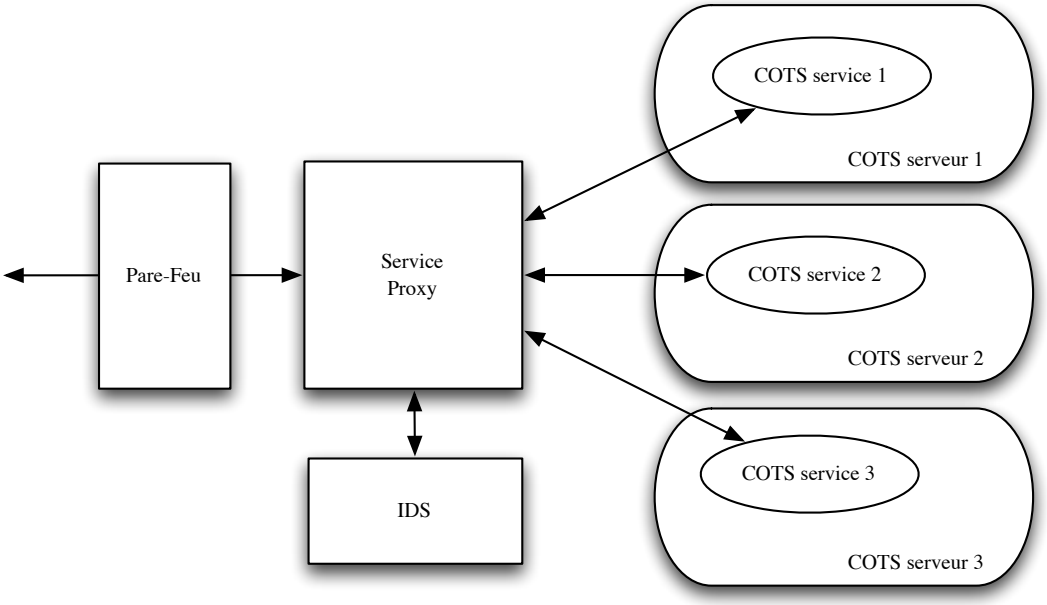


FIGURE 3.1 – Architecture Générale

3.2.3 Architecture pour la détection d'intrusion

L'architecture présentée Figure 3.1 est clairement inspirée des architectures classiques fondées sur la programmation N-Versions. Elle est composée de trois composants : un proxy, un IDS et un ensemble de serveurs diversifiés.

Le rôle du proxy est de prendre en entrée la requête envoyée au serveur, et de la faire suivre à chacune des variantes composant l'architecture. Il récupère ensuite les réponses fournies par chacune des variantes, et les envoie à l'IDS qui (1) sélectionne la sortie qui doit être fournie en réponse de la requête et (2) décide s'il y a lieu de lever une alerte compte tenu des réponses reçues des variantes.

Dans la pratique, assurer la détection d'une intrusion ne nécessite que deux serveurs. Toutefois, nous avons choisi d'utiliser trois serveurs pour assurer la tolérance à une intrusion. Dans cette architecture, la présence de trois serveurs permet d'assurer la continuité de service sous l'hypothèse qu'au plus une intrusion a lieu à un instant donné. Si l'un des trois serveurs fournit une réponse différente des autres, on peut considérer que le rôle de l'IDS est de lever une alerte, tout en diagnostiquant que le serveur défaillant est celui qui a fourni une réponse différente. Nous verrons par la suite que ce raisonnement n'est pas aussi simple dans le cas de l'utilisation de COTS.

Les trois serveurs diversifiés constituent le cœur de l'architecture : ils fournissent le

service demandé par le client. Idéalement, ces serveurs ne font pas seulement tourner un service diversifié : ils diversifient également le matériel et le système d'exploitation. L'objectif est clairement de réduire la probabilité de fautes de mode commun entre les différents serveurs, en assurant au maximum la décorrélation des fautes, et donc des vulnérabilités entre ces serveurs. Dans le cas idéal, nous pouvons ainsi supposer que l'hypothèse de ne voir qu'une vulnérabilité activée à un instant donné sur l'un des serveurs est vérifiée.

Suite à une intrusion, l'architecture est toujours capable d'assurer son service de manière correcte. La détection permet alors non seulement de lever une alerte, mais également de diagnostiquer le nœud défaillant et de le reconfigurer afin de passiver l'intrusion, et revenir dans une situation où l'on peut à nouveau tolérer une intrusion. Il faut noter ici que cette reconfiguration (par exemple, une restauration d'un état antérieur exempt d'intrusion) peut avoir lieu immédiatement après la détection d'une intrusion, ou périodiquement pour effectuer une réjuvénation du système (un retour à l'état de départ du système par exemple), ce qui peut se révéler utile dans le cas où une intrusion a eu lieu dans le système sans avoir été détectée.

3.2.4 Taxonomie des différences détectées

Dans la programmation N-Versions, une différence sur l'une ou plusieurs des sorties des variantes indique la présence de différences de conception dans certaines variantes. Comme toutes les variantes ont exactement la même spécification, ces différences sont en fait dues à des fautes résiduelles de conception. Dans notre cas particulier, les vulnérabilités font partie de ces fautes de conception. Toutefois, dans le cadre de la diversification de COTS, nous avons déjà expliqué que les différentes variantes peuvent avoir des spécifications différentes. Ceci a un impact non négligeable sur les causes des différences en sortie entre les différentes variantes.

Sur la figure 3.2, on voit que les différences entre les sorties des deux variantes différentes peuvent être dues :

- à des différences de spécification entre les variantes dans le cas où la définition des sorties pour certaines entrées ne fait pas partie de la spécification commune des variantes ;
- à des fautes de conception présentes dans les variantes lorsque la définition des sorties pour les entrées considérées font partie de la spécification commune des variantes. Parmi ces fautes de conception, on peut compter évidemment les vulnérabilités.

Dans notre approche, nous espérons pouvoir détecter des intrusions. Ainsi, nous nous intéressons tout particulièrement aux différences dues à l'exploitation de vulnérabilités. Ces vulnérabilités sont des fautes de conception : toutefois, elles se caractérisent différemment des fautes de conception *classiques*. En effet, on attend généralement d'une intrusion qu'elle viole la politique de sécurité du système, ce qui n'est pas le cas des autres fautes de conception. Ce qui signifie que les fautes de conception peuvent être

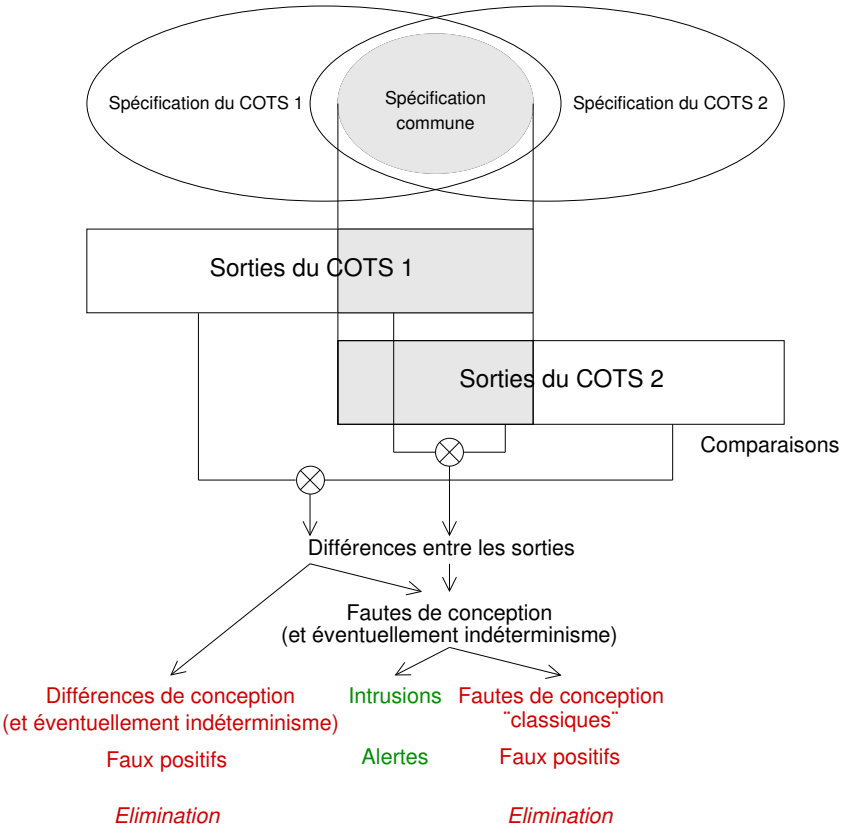


FIGURE 3.2 – Causes des différences détectées

divisées en deux classes : d'une part les vulnérabilités qui mènent ou peuvent mener à une violation de la politique de sécurité, et d'autre part les fautes de conception qui ne mènent pas à une telle violation. Distinguer les deux types de fautes automatiquement n'est toutefois pas une action facile à réaliser. En pratique, il faut l'expertise d'un administrateur humain pour réaliser cette tâche.

Compte tenu de notre objectif, les différences en sortie qui sont dues à des fautes de conception *classiques* ou à des différences de spécification conduisent en fait à des faux positifs. Ces faux positifs devraient, idéalement, être éliminés. Dans la pratique, les COTS que nous avons sélectionnés ont été utilisés depuis des années, et on peut raisonnablement penser qu'ils intègrent peu de fautes de conception *classiques* (cette hypothèse a été vérifiée par les tests qui ont été menés). Par conséquent, nous ne nous attendons pas à détecter de nombreuses différences dues à des fautes *classiques*. Ainsi les différences en sortie seront dues essentiellement à des différences de spécification ou à l'exploitation de vulnérabilités. Dans nos travaux, nous nous sommes focalisés sur l'élimination des différences en sorties dues à des différences de spécification entre les variantes. Dans ce but, nous avons mis en place des mécanismes de masquage de ces

différences qui se divisent en deux types de mécanismes distincts : les mécanismes de masquage préalables à l'envoi de la requête (ces mécanismes consistent à normaliser la requête pour qu'elle soit traitée de manière identique par toutes les variantes), et les mécanismes de masquage post-traitement (ces mécanismes cherchent à décider si, compte tenu des entrées, une différence en sortie est symptomatique d'une différence de spécifications ou non).

3.2.5 Mécanismes de masquage pour la diversification de COTS

Les mécanismes de tolérance aux fautes fondés sur la diversification fonctionnelle mettent généralement en place un mécanisme particulier de masquage d'erreurs : un vote. Dans le cadre de la programmation N-Version, l'hypothèse de décorrélation des fautes entre les variantes étant faite, on sait que si f fautes sont activées simultanément dans f différentes variantes, on obtient f sorties différentes de ces f variantes, et des sorties identiques pour toutes les autres variantes correctes. Si on a f+2 variantes, on obtient deux sorties identiques qui sont suffisantes pour savoir que ces sorties sont valides. Cette hypothèse est valide pour des fautes accidentelles. On peut maintenant se poser la question de sa validité dans le cas d'attaques : pour qu'un attaquant viole cette hypothèse, cela supposerait qu'avec la même entrée il réussisse à utiliser deux vulnérabilités différentes dans deux variantes différentes tout en ayant des sorties identiques pour chacune de ces variantes. Cela reste hautement improbable, même si cela est théoriquement possible. Nous supposerons donc par la suite que f+2 variantes suffisent à tolérer f intrusions contre le système diversifié, et que le mécanisme de vote a besoin de seulement deux valeurs identiques pour décider de la valeur correcte. Il faut noter ici que l'on pourrait relâcher l'hypothèse de décorrélation des fautes en supposant que f fautes corrélées peuvent être activées : dans ce cas, on devrait utiliser en sortie un vote majoritaire, ce qui nécessite, pour f vulnérabilités activées, d'avoir cette fois $2f + 1$ variantes.

Formellement, un mécanisme de vote dans ce contexte peut être décrit comme suit : soit I l'ensemble des entrées du service et I_{diff}, le sous-ensemble de I des entrées qui produisent une sortie incorrecte pour au moins une des variantes. Soit $O^i = \{o^i_1, o^i_2, ..., o^i_N\}$ l'ensemble des sorties des variantes pour une entrée $i \in I_{diff}$. Le vote V est une fonction de masquage $V : O^i \mapsto o^i_k$ qui retourne la valeur correcte du service si $\exists(k, l)/o^i_k = o^i_l$, ou une erreur sinon. Il faut noter que, vu les hypothèses, une seule valeur peut apparaître en plusieurs exemplaires (au moins 2), et que cette valeur est la valeur correcte.

Dans le cas de la diversification de COTS, les éléments de O^i peuvent être différents les uns des autres à cause d'une différence de spécification entre les versions, même si aucun faute n'a été activée. Ces différences doivent être définies comme légales, et doivent donc être également masquées. Afin de masquer ces différences, nous utilisons une relation d'équivalence : pour $i \in I_{diff}$, deux sorties $(o^i_j, o^i_k) \in O^i \times O^i$ sont équivalentes (ce qui est noté $o^i_j \equiv o^i_k$) si elles sont des sorties correctes connues des variantes j et k (c'est-à-dire qu'aucune intrusion n'a eu lieu). La fonction de vote V est une fonction

de masquage $V : O^i \mapsto o^i_k$ qui renvoie une valeur correcte $o^i_k \in O^i$ si $\exists(k,l)/o^i_k \equiv o^i_l$, ou une erreur sinon.

Ces deux définitions sont très proches en terme de complexité algorithmique. Par conséquent, la règle de masquage que nous définissons ci-après ne produit par un algorithme beaucoup plus complexe que le mécanisme de vote initial.

3.2.6 Tolérance aux intrusions aux niveaux des proxys et IDS

Si les serveurs diversifiés offrent une tolérance aux intrusions, ce n'est pas le cas des deux autres éléments de l'architecture, à savoir le proxy et l'IDS. Ces deux entités peuvent être directement attaquées par un utilisateur malveillant, sans qu'aucun mécanisme n'assure leur bon fonctionnement en présence d'intrusions. Toutefois, il faut noter que la complexité de ces deux éléments est relativement faible, de sorte qu'on peut raisonnablement espérer qu'ils soient complètement vérifiés. Il devient alors naturel de penser que ces deux éléments sont exempts de vulnérabilités, ou au moins qu'ils contiennent moins de vulnérabilités que les serveurs. Toutefois dans le cadre du projet ACI-SI DADDi, il a été envisagé d'utiliser une redondance passive sur ces deux éléments afin d'assurer leur tolérance aux intrusions.

3.3 Un IDS pour les serveurs web à base de diversification de COTS

L'approche que nous avons présentée dans la section précédente peut être appliquée à n'importe quel type de service (serveur ftp, serveur de courrier électronique, etc.) si l'hypothèse de la décorrélation des fautes (et des vulnérabilités dans notre cas) entre les serveurs diversifiés est vérifiée. Dans cette partie du document, nous appliquons l'approche à des serveurs web afin de démontrer la faisabilité de la méthode à travers des résultats expérimentaux.

Le choix de ce type d'application n'est pas anodin : les serveurs web constituent aujourd'hui le service indispensable à beaucoup d'entreprises, voire toutes. De plus en plus d'applications critiques sont développées sous forme de portails web dans de nombreux domaines, parmi lesquels le domaine bancaire, le commerce électronique, etc. Malheureusement, les serveurs web et les applications web sont devenus les cibles privilégiées des attaques sur internet. Pour preuve, Snort 2.2 détient 1064 signatures dédiées à des attaques web, pour un total de 2510 signatures. Enfin, de nombreux serveurs web sont disponibles sous formes de COTS, permettant une diversification du service.

Dans cette section, nous décrivons l'algorithme de détection dans le cas de serveurs web, et discutons des choix que nous avons faits. Ensuite, nous donnons différents résultats expérimentaux qui montrent la pertinence et l'efficacité de l'approche.

3.3.1 Algorithme de détection

Bien que l'approche soit générale, l'algorithme de détection est spécifique au type d'application diversifiée. Dans ce paragraphe, nous décrivons l'algorithme que nous avons mis en place.

Nous considérons les serveurs web comme des boites noires, c'est-à-dire que nous ne pouvons observer que leurs échanges avec leur environnement (en fait leurs entrées et leurs sorties). La partie de la spécification commune à chacun des COTS est la spécification du protocole HTTP. La spécification commune ne décrit a priori pas d'autres sorties, comme des appels systèmes réalisés sur les systèmes hôtes.

Le protocole HTTP se fonde sur un schéma de requête/réponse. Un client établit une connexion avec le serveur et envoie une requête. Une réponse est donnée en retour par le serveur. Cette réponse est composée de deux parties distinctes : l'entête (*header*) et le corps du message (*body*). La partie entête est parfaitement définie. Par exemple, le premier champ d'entête décrit le statut, qui est composé de la version du protocole HTTP et le code de statut de la requête. Le corps du message peut au contraire contenir n'importe quel type d'information, en fonction de la requête envoyée.

Une comparaison bit à bit des entêtes HTTP ne peut être réalisée directement, car différents serveurs peuvent utiliser différents entêtes. De plus certains entêtes peuvent être renseignés de façon différente par les différentes variantes. On peut prendre en exemple les champs *Date* et *Server*. La sémantique de chaque entête doit être prise en compte dans le processus de comparaison. Ceci permet de conclure qu'il est nécessaire de faire un choix des entêtes à comparer dans l'algorithme de détection. Le code de statut peut par exemple faire partie de ces choix. D'autres choix peuvent être intéressants, tels que les champs *Content-Length*, *Content-Type*, *Last-Modified*. Ces trois entêtes sont en général envoyés par tous les serveurs web. De plus une différence sur ces champs peut indiquer une attaque contre l'intégrité des fichiers utilisés par les serveurs web.

Les corps des messages, lorsqu'ils ne sont pas vides, peuvent également être comparés. Dans notre approche, nous nous sommes restreints à des sites web statiques. Dans ce contexte, le fichier renvoyé par les différents serveurs pour une requête donnée devrait être identique, et les corps de réponses peuvent être comparés bit-à-bit. Toutefois, même avec cette restriction, la comparaison binaire du corps des messages peut poser problème : par exemple, le contenu d'un répertoire identique renvoyé par des serveurs différents peut mener à des corps de messages différents (ce qui peut clairement être défini comme une différence de spécification entre les variantes). Dans le cadre de nos expérimentations, nous avons désactivé pour toutes les variantes leur capacité de construire dynamiquement ce type de contenu.

L'algorithme de détection est composé de deux phases distinctes :
– un *watchdog timer* permet de détecter une défaillance par arrêt de l'un des ser-

Algorithm 1: Algorithme de détection d'intrusion par comparaison de réponses

Data: n le nombre de serveurs web et $R = \{R_i | 1 \leq i \leq n\}$ l'ensemble des réponses des serveurs web pour une requête donnée Req, D l'ensemble des différences de conception connues pour les serveurs web, $headers$ le tableau défini par $headers[0]$ = 'Content-Length', $headers[1]$ = 'Content-Type', $headers[2]$ = 'Last-Modified'

if $(Req, R) \in D$ **then**
  /* Prise en compte des différences de conception qui ne sont pas des vulnérabilités */ ;
  modifier R pour masquer les différences ;

Partitionner R in C_i, $\forall (R_l, R_k) \in C_i^2, R_l.statusCode = R_k.statusCode$
$1 \leq i \leq m \leq n$ $\forall (i,j) \in [1,m]^2, i \neq j, C_i \cap C_j = \emptyset$, m le nombre de partitions

if $\forall i\ Card(C_i) < 2$ **then**
  Lever une alerte /* Pas de réponse correcte */ ;
  Exit ;
else
  /* 2 réponses correctes existent dans l'ensemble des réponses */ ;
  Trouver p tel que $Card(C_p) \geq 2$;
  if $C_p.statusCode \neq 2XX$ **then**
    /* Pas de réponse correcte : pas besoin de regarder d'autres entêtes */ ;
    for $k = 1$ *to* n **do**
      if $R_k \notin C_p$ **then** Lever une alerte concernant le serveur k;
    Exit ;
  else
    /* Un ensemble de réponses correctes nécessite une investigation */ ;
    for $i = 0$ *to* 2 **do**
      /* Comparer les entêtes */ ;
      $T \longleftarrow C_p$ /* Nous affectons à T l'ensemble des réponses identiques */ ;
      Vider tous les C_j /* Nous effaçons les partitions avant d'en créer de nouvelles */;
      Partitionner T en C_j, $\forall (R_l, R_k) \in C_j^2, R_l.headers[i] = R_k.headers[i]$
      $1 \leq j \leq m \leq Card(T)$ $\forall (j,k) \in [1,m]^2, j \neq k, C_j \cap C_k = \emptyset$, m le nombre de partitions
      if $\forall j\ Card(C_j) < 2$ **then**
        Lever une alerte /* Aucune réponse correcte n'a été trouvée */ ;
        Exit ;
      else
        /* Plus de 2 réponses identiques ont été trouvées */ ;
        Trouver p tel que $Card(C_p) \geq 2$;
    /* Comparer les corps (bodies) */ ;
    $T \longleftarrow C_p$;
    Vider tous les C_i;
    Partitionner T in C_j, $\forall (R_l, R_k) \in C_j^2, R_l.body = R_k.body$
    $1 \leq j \leq m \leq Card(T)$ $\forall (i,j) \in [1,m]^2, i \neq j, C_i \cap C_j = \emptyset$, m le nombre de partitions
    if $\forall j\ Card(C_j) < 2$ **then**
      Lever une alerte /* Aucune réponse correct n'a été trouvée */ ;
      Exit ;
    else
      /* Au moins deux réponses identiques ont été trouvées */ ;
      trouver p tel que $Card(C_p) \geq 2$;
      for $k = 1$ *to* n **do**
        if $R_k \notin C_p$ **then**
          /* la réponse k n'est pas correcte */ ;
          Lever une alerte concernant le serveur k;
      Exit ;

veurs. A la fin du temps d'attente défini par le timer, toute variante n'ayant pas répondu est considérée comme défaillante ;
– toutes les réponses reçues dans le temps imparti sont ensuite utilisées dans l'algorithme de détection.

L'algorithme 1 donne tous les détails du processus de comparaison dans le cas de serveurs web. Lorsque toutes les réponses des serveurs sont collectées, nous commençons par identifier si ces réponses correspondent à des différences de spécification connues entre les serveurs. Si c'est un cas avéré, nous modifions les réponses (en particulier les entêtes) pour les uniformiser, ce qui revient à masquer ces différences. Nous pouvons alors appliquer l'algorithme de comparaison en lui-même. Comme la comparaison du corps de message peut être consommatrice de temps, nous commençons par comparer les valeurs des entêtes retournées par les différentes variantes. Dans l'ordre, nous comparons le code de statut, puis les champs *Content-Length*, *Content-Type*, et *Last-Modified*, et enfin le corps des réponses. Si au moins deux réponses identiques ne peuvent être trouvées successivement pour chacun de ces comparaisons, l'algorithme s'arrête et lève une alerte. Il faut noter qu'il est inutile de comparer les corps des réponses et les autres entêtes si le code de statut n'est pas du type 2XX (c'est-à-dire que la requête n'a pas été exécutée avec succès). En effet, dans ce cas précis, la réponse est générée dynamiquement et peut différer d'un serveur à l'autre, ce qui mènerait à de fausses alertes.

Masquage de différences en sortie Il est utile ici de préciser comment on peut reconnaître des différences *normales* entre les sorties des différentes variantes. Dans notre approche, cette reconnaissance est réalisée à l'aide de règles qui définissent ces comportement normaux. En fait, ces règles mettent en relation différents paramètres comme par exemple une caractéristique de la requête (sa longueur, son adéquation avec une expression régulière, etc.), le code de statut de la réponse, et des entêtes *Content-type*. Par exemple, une règle peut définir une relation entre les sorties, par exemple les codes de statut des différentes sorties. Un autre exemple pourrait être de définir un lien entre une entrée d'un certain type avec les sorties attendues. Ces règles mettent en place la relation d'équivalence que nous avons définie dans le paragraphe 3.2.5. Afin d'illustrer l'approche, la figure 3.3 décrit une règle particulière où nous définissons que le serveur Apache ne renvoie pas le même code de statut que IIS (sous Windows) ou thttpd (sous Linux) lorsqu'un contenu de répertoire est demandé mais que le dernier caractère '/' est manquant dans la requête : Apache retourne le code de statut 301 quand IIS et thttpd renvoie 302. Lorsque cette différence connue est détectée, le code de statut de la réponse d'Apache est modifié pour être égal à celui des deux autres serveurs.

La définition de ces règles peut paraître difficile pour plusieurs raisons : la première est qu'il est nécessaire de les écrire manuellement, ce qui demande une connaissance approfondie du fonctionnement de plusieurs serveurs. En outre la base de règles doit être mise à jour à chaque changement de versions des serveurs web. La deuxième est que chaque règle doit être suffisamment précise pour correctement identifier une différence de spécification, sans pour autant masquer des comportements en présence d'attaques.

```
<request id="0">
  <description>directory request without a final '/'</description>
  <regexpURI>^.*[^/]$</regexpURI>
  <regexpMethod>^.*$</regexpMethod>
  <serverBehaviour>
    <server>
      <name>apache</name>
      <httpcode>301</httpcode>
      <contentType>text/html</contentType>
    </server>
    <server>
      <name>iis</name>
      <name>thttpd</name>
      <httpcode>302</httpcode>
      <contentType>text/html</contentType>
    </server>
  </serverBehaviour>
  <compareContent>no</compareContent>
  <response>iis</response>
  <warn>no</warn>
  <alert>no</alert>
</request>
```

FIGURE 3.3 – Directory Rule : Les tags regexpURI (definition de l'URL) et regexpMethod (definition de la méthode HTTP) définissent les entrées des serveurs en utilisant des expressions régulières. Les tags server définissent les sorties des servers S_i, c'est-à-dire les sorties attendues produites par les différents serveurs quand ils prennent les valeurs définies en entrées. Les autres tags : compareContent, response, warn et alert définissent l'ensemble des actions qui doivent être réalisées par l'IDS afin de masquer la différence ou générer une alerte.

Dans la pratique, nous avons construit notre base de règles en analysant les alertes levées par l'IDS sur un mois de logs. Ce travail ne nous a pris qu'une seule journée. Cet effort est donc tout à fait raisonnable.

Il faut noter ici que la méthode décrite ci-dessus ne permet pas de masquer toutes les différences normales entre les serveurs. Par exemple, Windows ne différencie pas les lettres majuscules des lettres minuscules dans une URL, contrairement aux autres systèmes utilisés. Cette différence mène à de nombreuses fausses alertes. Le meilleur moyen de traiter cette différence est de normaliser les requêtes avant leur exécution par les serveurs, de sorte qu'ils se comportent tous de la même manière.

Par conséquent, les mécanismes de masquage se divisent en deux classes distinctes : les mécanismes appliqués préalablement à l'exécution de la requête, et qui consistent à standardiser les entrées, et les mécanismes postérieurs à l'exécution de la requête qui

masquent les différences en sortie qui ne sont pas dues à des attaques.

3.3.2 Résultats expérimentaux

L'objectif des tests qui ont été menés est de permettre de savoir si l'IDS proposé détecte bien des attaques connues d'une part, et d'évaluer les faux positifs générés d'autre part.

Attaques contre...	BuggyHTTP	IIS	Apache
Confidentialité	Lecture du fichier /etc/shadow	CVE-2000-0884	CAN-2001-0925
Intégrité	Modification du fichier /etc/shadows	CVE-2000-0884	-
Disponibilité	Défaillance par arrêt du serveur	CVE-2000-0884	-

TABLE 3.1 – Attaques contre les serveurs web

Détection d'attaques connues Dans la première phase de validation, nous avons utilisé un environnement constitué de trois serveurs : BuggyHTTP [1] sur Linux, Apache 1.3.29 sur MacOS-X et IIS 5.0 sous Windows. Les attaques qui ont été menées sont résumées dans le tableau 3.1. Toutes les attaques menées ont été détectées par l'IDS.

Comportement de détection en environnement réel Dans cette deuxième phase de test, nous avons utilisé un environnement composé de trois serveurs : un serveur Apache sur MacOS-X, un serveur thttpd sur Linux, et IIS 5.0 sous Windows. Les trois serveurs contiennent une copie du site web de Supelec Rennes. Les trois serveurs sont alimentés par les requêtes réelles ayant eu lieu sur le serveur de Supelec pendant la période de mars 2003. Cela représente plus de 800.000 requêtes. Comme le montre la figure 3.4, seulement 0.37% des différences en sortie mènent à la levée d'une alerte, ce qui représente 0.016% des requêtes HTTP pour un total de 131 alertes dans le mois. Ce qui est relativement peu. De plus si on analyse ces alertes (cf. figure 3.5), on s'aperçoit que seuls les quatre premiers types d'alertes sont probablement des faux positifs (22% des alertes). Les trois premières requêtes de type *winsys access* sont par contre de vraies attaques contre le serveur IIS.

Il faut noter ici que les mécanismes de masquage ont permis de masquer 99,63% de différences qui ne sont pas dues à des attaques. Ces masquages sont largement dus aux règles appliquées postérieurement au traitement de la requête. Il faut noter que pour ces tests, la base de règles ne contenait que 36 règles, ce qui est relativement peu. Nous

1. BuggyHTTP est un petit serveur jouet contenant intentionnellement de nombreuses vulnérabilités

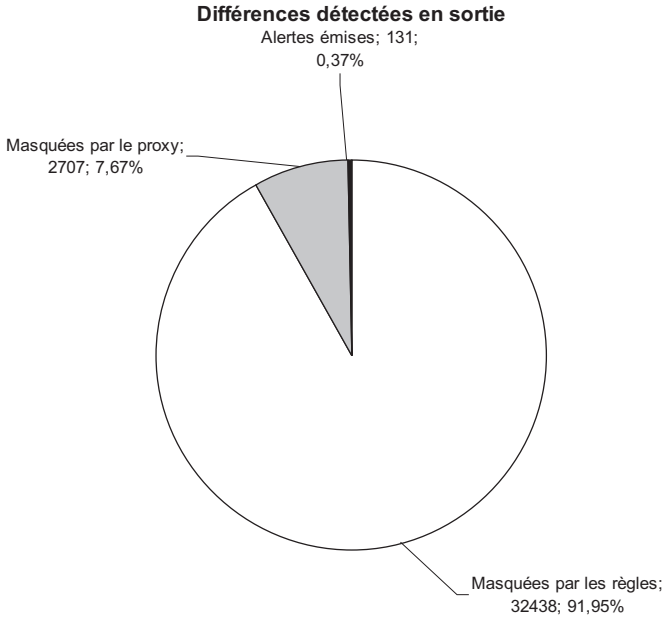


FIGURE 3.4 – Analyse des différences détectées

pensons d'ailleurs que cette base restera toujours de taille très limitée, ce qui reflète le fait que les différences de spécification entre les serveurs sont en nombre réduit.

Comparaison avec Snort et WebStat On peut d'autre part se comparer avec d'autres IDS. Nous avons envisagé de nous comparer à deux autres IDS très connus, l'un générique (Snort) et l'autre spécialiste de la détection des attaques web (WebStat [VRKK03]). Il faut toutefois être prudent dans ce processus de comparaison : en effet ces trois IDS ne détectent pas les mêmes types d'attaques, car ils ne partagent pas la même base de signature. De plus ils ne génèrent pas forcément les mêmes faux positifs.

Nous avons fourni aux trois IDS le même trafic que précédemment. La figure 3.6 montre que généralement Snort et WebStat produisent environ 10 alertes par jour, tandis que notre IDS en produit 5. Ceci peut être expliqué par le fait que Snort et WebStat détectent des attaques (c'est-à-dire des tentatives d'intrusion), alors que notre IDS détecte, lui, des intrusions (une alerte est symptomatique d'une attaque probablement réussie). On peut facilement conclure à la lecture de ces chiffres que notre IDS produit moins d'alertes que les deux autres IDS choisis, et par conséquent moins de faux positifs. Ceci doit toutefois être tempéré par le fait que Snort et WebStat ont été utilisés dans leur configuration standard, sans réellement les optimiser pour notre situation de test, ce qui peut faire baisser leurs performances.

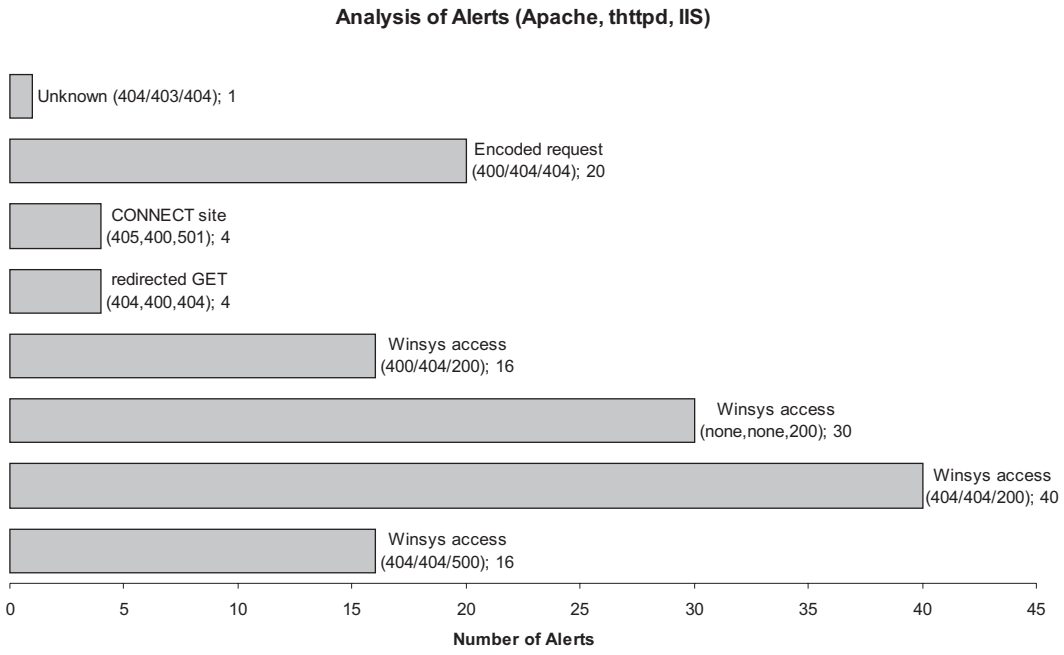


FIGURE 3.5 – Analyse des alertes levées

3.4 Positionnement des travaux

L'utilisation de la diversification de COTS pour la tolérance aux intrusions ou la détection d'intrusion a été étudiée dans quelques projets. Nous en citons deux ici : le projet DIT d'une part et le projet HACQIT d'autre part.

3.4.1 Projet DIT

Le projet DIT (Dependable Intrusion Tolerance) est un projet qui propose une architecture générale pour réaliser des systèmes industriels tolérants aux intrusions [SND08]. L'approche proposée par le projet est illustrée dans le cadre de serveurs web tolérants aux intrusions. L'architecture se compose d'un ensemble de serveurs proxy redondants qui transfèrent les requêtes à un ensemble de serveurs applicatifs diversifiés. La diversification est ici un moyen pour tolérer les intrusions, et ne sert pas immédiatement à la détection, même si cette possibilité est évoquée (en comparant un *hash* MD5 des réponses des différentes variantes). En fait la configuration entière du système est surveillée par divers mécanismes vérifiant l'intégrité des serveurs dont un système de détection d'intrusion basé sur le framework EMERALD. Un système de gestion distribuée est présent pour décider de la politique de tolérance aux intrusions à adopter par le système en fonction des conditions externes. Ce système peut en outre décider d'utiliser des protocoles d'accord plus stricts (entre les répliques des proxys), de filtrer

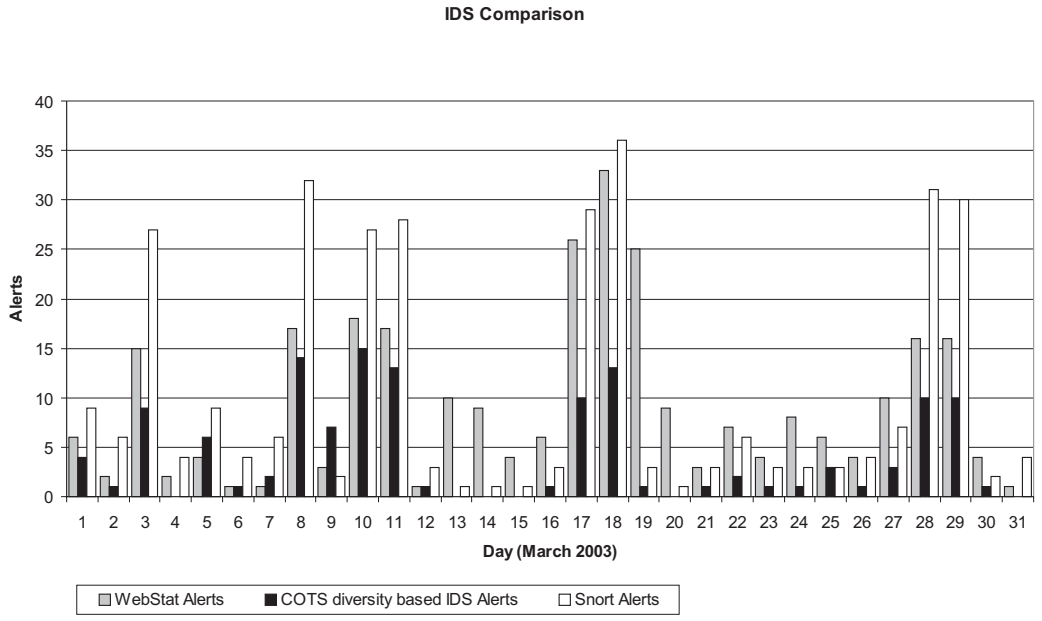


FIGURE 3.6 – Comparaison avec Snort et WebStat

les requêtes provenant de clients suspects ou encore de redémarrer des serveurs ou des services qui sont compromis.

3.4.2 Projet HACQIT

Le projet HACQIT [JRC+02] propose un système tolérant aux intrusions capable d'apprendre la signature de nouvelles attaques. La capacité de détection est fondée sur la comparaison des résultats de deux serveurs web COTS diversifiés (un serveur IIS et un serveur Apache). Une différence entre les deux réponses signifie la présence d'une attaque, et la nécessité de construire la signature d'une nouvelle attaque si celle-ci n'est pas déjà connue. La construction de cette signature se fonde sur l'utilisation d'outils de monitoring du système (Snort, moniteur hôtes, contrôlée d'intégrité, moniteur applicatif, etc.). Dans ces travaux, la détection des différences entre les réponses des serveurs est assez restreinte : si les deux serveurs répondent, elle se focalise sur le code de statut HTTP de la réponse, et deux cas seulement sont pris en compte (combinaison de code de retours 2XX/4XX [réussite et erreur du client] et 2XX/3XX [réussite et redirection]). L'un des inconvénients de l'approche est que la présence de deux serveurs ne permet pas de diagnostiquer aisément lequel des deux serveurs est compromis en cas de différence sur les sorties.

3.5 Conclusion

Bien que la diversification fonctionnelle ait déjà été étudiée dans d'autres projets, nous avons fourni les seuls résultats expérimentaux publiés dans le cadre de la détection d'intrusion. On remarque que les attaques que nous avons jouées ont été détectées, et que les mécanismes proposés induisent peu de faux positifs, grâce à des mécanismes de masquage qui se révèlent donc indispensables dans cette situation. Dans cette approche, toute attaque qui a une influence sur les sorties des variantes utilisées est détectée. Ce qui signifie qu'une attaque ayant pour conséquence l'arrêt d'un serveur est détectée. De même toute attaque visant à récupérer des données confidentielles sur un serveur (données qui passent par la réponse du serveur) est détectée.

Toutefois, nous devons ici émettre un bémol. Une attaque contre l'intégrité d'un des serveurs, et qui n'aurait pas de conséquence sur la réponse ne pourrait pas être détectée par l'approche. Afin de détecter une telle attaque, il faudrait avoir une connaissance interne du fonctionnement du serveur attaqué. Cette remarque a motivé les travaux qui sont présentés dans le chapitre suivant.

Chapitre 4

Détection d'intrusion par comparaison de graphes de flux d'information

4.1 Introduction

Lorsqu'on met en place une architecture à base de diversification fonctionnelle, il convient, pour tolérer les fautes, d'implémenter un vote sur les sorties des différentes variantes de l'architecture. Dans le chapitre 3, nous avons choisi de comparer les entrées-sorties réseau de chaque serveur HTTP, c'est-à-dire que nous avons considéré les variantes comme des boîtes noires. Cette approche a l'avantage de n'avoir besoin d'aucune information complémentaire, par exemple relative au fonctionnement interne de chacune des variantes sur les différents serveurs.

Toutefois, il serait possible de déplacer le point de comparaison aux interactions entre le service et le système d'exploitation qui l'héberge, c'est-à-dire ne pas considérer les sorties seulement d'un point de vue réseau, mais également les sorties locales de chaque variante. L'inconvénient d'une telle approche est que les systèmes d'exploitation ne proposent pas les mêmes interfaces, c'est-à-dire que les appels systèmes appelés par chacune des variantes sont différents d'un serveur à l'autre. Toutefois, les actions réalisées par ces appels systèmes différents sont sensiblement les mêmes : ouverture/fermeture/lecture/écriture d'un fichier, communication entre processus, etc.

On peut caractériser chaque appel système par l'information qu'il véhicule entre le service observé et le système d'exploitation (le contenu d'un fichier par exemple), et il est donc possible d'observer le comportement du service diversifié en modélisant les flux d'information qu'il produit durant l'exécution d'une requête. Ces flux d'information constituent les entrées-sorties observables sur chacun des serveurs diversifiés, et peuvent donc être comparés d'un serveur à l'autre. Le réel apport de cette approche est, qu'en cas de différence de comportement entre deux serveurs, cette différence peut

être mise en évidence, apportant ainsi des capacités de diagnostic qui manquaient à l'approche en boite noire du chapitre précédent.

Dans ce chapitre, nous allons présenter une approche originale de comparaison des comportements des variantes d'un service diversifié en comparant les graphes de flux d'information générés sur chaque serveur par la variante locale. Cette comparaison se fera en calculant une similarité entre graphes, grâce à laquelle nous allons pouvoir détecter des déviations de comportement, et donc des intrusions. Dans la section 4.2, nous définissons notre méthode de détection, puis nous montrons la capacité de détection de cette méthode sur une étude expérimentale section 4.3 et enfin un avantage qu'elle présente en section 4.4, à savoir des capacités de diagnostic.

Ces travaux ont été publiés dans IFIP SEC 2008 [MTMS08].

4.2 Détection d'intrusion par comparaison de graphes de flux d'information

Nous nous plaçons dans ce chapitre dans la même situation que le chapitre précédent. Nous possédons une architecture diversifiée composée de COTS (cf. la figure 3.1). Ces COTS sont des serveurs web. Sur chacun des serveurs diversifiés, nous construisons dynamiquement le modèle qui devra être comparé à celui obtenu sur les autres serveurs. Afin de décrire ces modèles, nous devons tout d'abord définir la notion de graphe de flux d'information. Puis nous donnons une définition de la notion de similarité entre deux graphes de flux d'information, et montrons comment cette similarité peut être utilisée pour détecter des intrusions.

4.2.1 Graphes de flux d'information

La notion de flux d'information que nous utilisons est similaire à celle utilisée par Bell et Lapadula [BL76]. Ils définissent deux types d'entités dans le système : les entités actives, appelées sujets (tels que les utilisateurs ou les processus), et les entités passives, les objets, qui sont les détenteurs de l'information (les fichiers, les sockets). Dans ce modèle, les flux d'information peuvent être produits de trois façons différentes :

- un sujet peut lire un objet sous certaines conditions, ce qui a pour effet de produire un flux d'information de l'objet vers le sujet ;
- un sujet peut écrire dans un objet sous certaines conditions, ce qui a pour effet de produire un flux d'information du sujet vers l'objet ;
- un sujet s_1 peut invoquer un sujet s_2 (création de processus, communication entre processus), ce qui a pour conséquence de produire un flux d'information de s_1 vers s_2.

Les appels systèmes qui permettent de réaliser ces opérations diffèrent d'un système à l'autre, mais peuvent toujours se ramener à l'une de ces trois catégories.

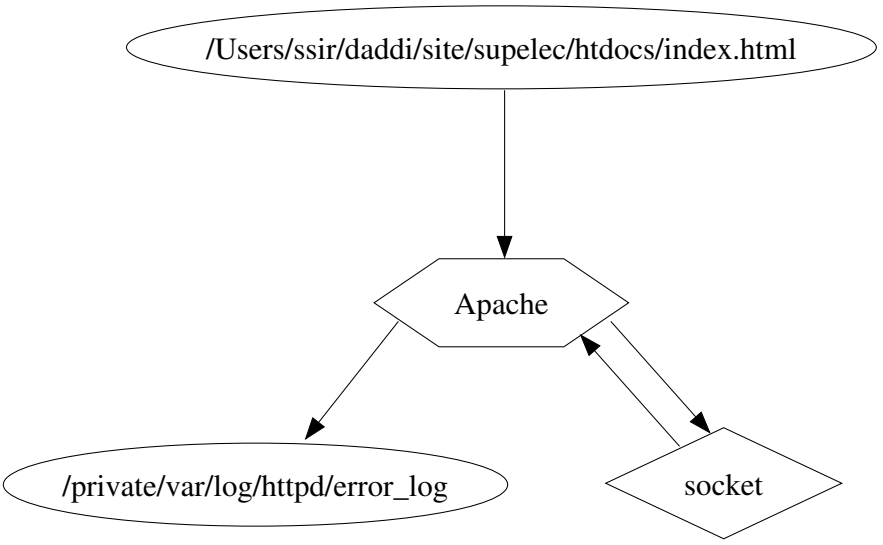


FIGURE 4.1 – Graphe de flux d'information pour une requête HTTP traitée par le serveur Apache sous MacOS-X

Nous appelons *graphe de flux d'information* un ensemble de flux d'information, ainsi que d'objets et de sujets impliqués dans la création de ces flux, lors d'un appel au service surveillé. Un graphe de flux d'information est un graphe orienté étiqueté, c'est-à-dire que les nœuds et les arcs sont associés à des étiquettes. Les sujets et objets du système d'exploitation (processus, fichiers, sockets, ...) sont les nœuds du graphe et les flux d'information les arcs du graphe. Les étiquettes sont des informations reliées aux noeuds ou aux arcs du graphe. Dans notre prototype, les étiquettes que nous avons définies sont les suivantes :

- chaque nœud est associé à un des types suivants : PROCESS, SOCKET, FILE, PIPE, MAPPING ;
- un arc est également associé à un type parmi : PROCESS_TO_FILE, FILE_TO_PROCESS, PROCESS_TO_SOCKET, SOCKET_TO_PROCESS, PROCESS_TO_PIPE, PIPE_TO_PROCESS, PROCESS_TO_LOGFILE.

D'autres données sont associées aux nœuds ou aux arcs, mais ne sont pas considérées comme des étiquettes, c'est-à-dire qu'elles ne sont pas prises en compte dans le calcul de la similarité, telle que définie dans la section suivante. Les nœuds de type FILE sont associés à un nom, un descripteur de fichier, une date de création et de destruction du descripteur de fichier dans le système d'exploitation. Les processus sont associés à leur nom, leur *pid* (numéro d'identification du processus dans le système), le *pid* de leur parent et la date de création et de destruction du processus dans le système. Les arcs sont associés avec les données transférées entre la source et la destination du flux d'information et avec un temps d'accès.

La figure 4.1 présente un exemple de graphe de flux d'information. Ce graphe de flux est le résultat de l'observation au niveau du système de l'exécution d'une requête HTTP par le serveur Apache tournant sous MacOS-X. La forme des nœuds représente le type des objets : les sockets sont représentées par des losanges, les processus sont représentés par des hexagones, les fichiers sont représentés par des ellipses. Le graphe montre que le processus Apache lit une socket (sur laquelle il reçoit la requête), lit le fichier correspondant à la requête (ici le fichier *index.html*), écrit dans le fichier de log *access.log* et écrit sur la socket la réponse renvoyée par le serveur. Il faut noter que la chronologie des flux d'information est donnée par les temps d'accès associés aux flux, temps qui ne sont pas représentés sur la figure.

4.2.2 Similarité de graphes de flux d'information

Afin de comparer deux à deux les graphes produits par deux exécutions de requêtes sur deux serveurs différents, nous avons décidé d'effectuer un calcul de similarité entre ces graphes. Pour cela, nous avons décidé d'utiliser le modèle développé par Champin et Solnon [CS03], qui permet de calculer la similarité entre deux graphes étiquetés. Dans ces travaux, ils proposent un algorithme pour calculer la similarité entre deux graphes, algorithme que nous avons légèrement optimisé pour notre cas particulier. Nous allons présenter brièvement leur approche et détailler l'algorithme que nous utilisons pour comparer des graphes de flux d'information.

Formellement, un graphe étiqueté est un graphe orienté $G = (V, r_V, r_E)$ où :
- V est l'ensemble des nœuds ;
- $r_V \subseteq V \times L_V$ est une relation entre les nœuds du graphe et les étiquettes des nœuds (L_V est l'ensemble des étiquettes des nœuds) ;
- $r_E \subseteq V \times V \times L_E$ est une relation entre les arcs et les étiquettes des arcs (L_E est l'ensemble des étiquettes des arcs).

Le descripteur d'un graphe étiqueté $G = (V, r_V, r_E)$ est défini par $desc(G) = r_V \cup r_E$.

Considérons deux graphes étiquetés $G_1 = (V_1, r_{V_1}, r_{E_1})$ et $G_1 = (V_2, r_{V_2}, r_{E_2})$; nous cherchons à mesurer la similarité entre ces deux graphes. Pour cela, la notion de *mapping* est introduite : un *mapping* m est une relation $m \subseteq V_1 \times V_2$. m est un ensemble de couples de nœuds. Dans un *mapping* m, un nœud $v_1 \in V_1$ (resp. $v_2 \in V_2$) peut être associé avec zéro, un ou plusieurs nœuds de V_2 (resp. V_1). Une notation fonctionnelle peut être utilisée pour m : $m(v)$ est l'ensemble des nœuds qui sont associés à v dans le *mapping* m.

La similarité entre G_1 et G_2 en fonction d'un *mapping* m est alors donnée par la formule suivante :

$$sim_m(G_1, G_2) = \frac{f(desc(G_1) \sqcap_m desc(G_2))}{f(desc(G_1) \cup desc(G_2))}$$

où f est une fonction positive croissante en fonction de l'inclusion ensembliste (le cardinal est une fonction qui respecte ces critères par exemple) et $\sqcap_m$, appelé intersection en

fonction du *mapping* m, représente l'ensemble des éléments des descripteurs des deux graphes pour lesquels les étiquettes correspondent dans le *mapping* m :

$$\begin{aligned} desc(G_1) \sqcap_m desc(G_2) &= \{(v,l) \in r_{V_1} | \exists v' \in m(v), (v',l) \in r_{V_2}\} \\ &\cup \{(v,l) \in r_{V_2} | \exists v' \in m(v), (v',l) \in r_{V_1}\} \\ &\cup \{(v_i,v_j,l) \in r_{E_1} | \exists v'_i \in m(v_i), \exists v'_j \in m(v_j)(v'_i,v'_j,l) \in r_{E_2}\} \\ &\cup \{(v_i,v_j,l) \in r_{E_2} | \exists v'_i \in m(v_i), \exists v'_j \in m(v_j)(v'_i,v'_j,l) \in r_{E_1}\} \end{aligned}$$

Il est nécessaire d'introduire un dernier concept pour calculer la similarité entre deux graphes. Un nœud peut en effet être mappé avec plusieurs autres nœuds. Ce concept est intéressant dans notre cas car il permet d'associer plusieurs processus sur un OS avec un seul processus sur un autre OS par exemple. La notion de *splits* peut être ajoutée pour prendre en compte l'association d'un nœud particulier avec plusieurs autres nœuds. Les *splits* d'un *mapping* m représentent les nœuds qui sont associés avec strictement plus d'un nœud dans le *mapping* m :

$$splits(m) = \{(v, s_v) | v \in V_1 \cup V_2, s_v = m(v), |m(v)| \geq 2\}$$

La définition de la similarité en fonction d'un *mapping* m est alors modifiée :

$$sim_m(G_1, G_2) = \frac{f(desc(G_1) \sqcap_m desc(G_2)) - g(splits(m))}{f(desc(G_1) \cup desc(G_2))}$$

où g est une fonction positive, croissante en fonction de l'inclusion ensembliste. A partir de la définition de la similarité entre deux graphes en fonction d'un *mapping* m, il est possible de définir la similarité entre deux graphes :

$$sim(G_1, G_2) = \max_{m \subseteq V_1 \times V_2} \frac{f(desc(G_1) \sqcap_m desc(G_2)) - g(splits(m))}{f(desc(G_1) \cup desc(G_2))}$$

Trouver la similarité entre deux graphes, c'est trouver le *mapping* m qui maximise cette valeur. Les figures 4.2 et 4.3 donnent des exemples de *mappings*.

Champin et Solnon [CS03] ont proposé deux algorithmes pour résoudre ce problème : un algorithme de recherche complète avec quelques optimisations pour couper les branches de l'arbre de recherche et un algorithme glouton. Dans notre prototype, nous avons choisi d'utiliser l'algorithme de recherche complète. Nous sommes en effet plus intéressés par la précision du calcul de la similarité que par la rapidité de l'algorithme.

De plus, nous avons introduit certaines optimisations adaptées aux cas des graphes de flux d'information et plus particulièrement des serveurs web pour lesquels nous avons développé notre prototype :

- cela n'a pas de sens d'associer un processus à un fichier ou une socket ; ces *mappings* sont donc interdits ;

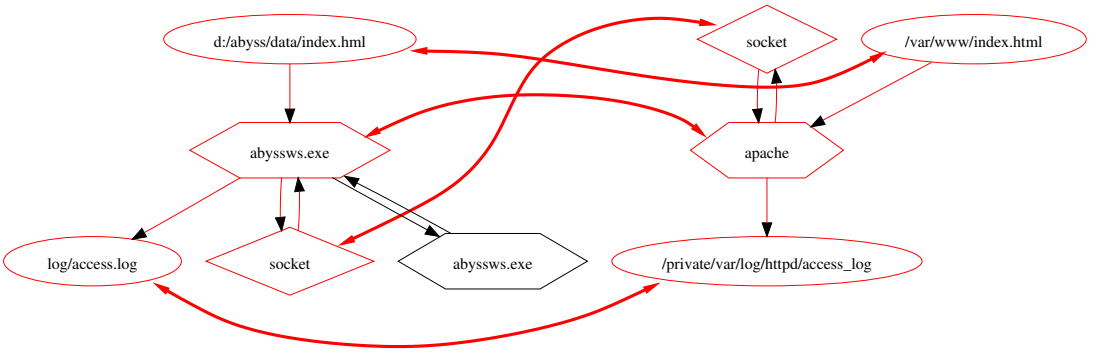


FIGURE 4.2 – Exemple de *mapping* entre deux graphes entrainant une similarité forte

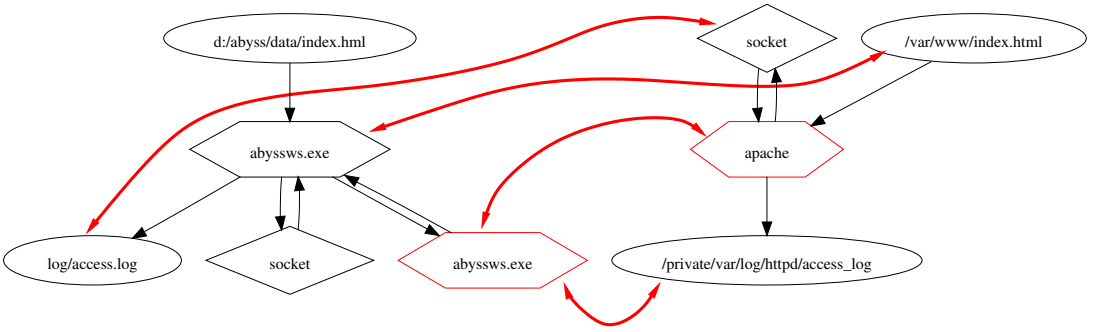


FIGURE 4.3 – Exemple de *mapping* entre deux graphes entrainant une similarité faible

- les communications entre processus par l'intermédiaire de fichiers ne sont pas présentes dans le cas particulier que nous considérons ; les associations entre une socket et un fichier ne sont pas permises dans un *mapping* (nous considérons que les fichiers ne sont pas des moyens de communications entre processus) ;
- un fichier en dehors de l'espace web et un fichier à l'intérieur de l'espace web ne peuvent être associés dans un *mapping*. De plus, à l'intérieur de l'espace web, seuls des fichiers possédant le même nom peuvent être associés dans un *mapping*.
- notre prototype étant développé pour les serveurs web, les sockets recevant les requêtes sont *mappées* ensemble au début de l'algorithme, ainsi que les processus lisant ces sockets.

Avec ces optimisations, le temps de calcul des *mappings* est relativement bon pour des graphes de petite taille. Puisque l'algorithme est exponentiel en temps en fonction du nombre de nœuds, le temps de calcul peut être relativement long pour des graphes plus importants : de l'ordre de 10 minutes pour des graphes de 15 nœuds par exemple. Notre prototype ne peut donc être utilisé en temps réel actuellement. Il serait, pour cela, nécessaire d'évaluer d'autres algorithmes tels que l'algorithme glouton proposé par Champin et Solnon et d'optimiser le code de notre prototype.

4.2.3 Détection d'intrusion

Les serveurs COTS implémentant le même service, les graphes de flux d'information doivent être sensiblement les mêmes sur les différents serveurs et donc la similarité doit être élevée dans les cas normaux (cela a été vérifié expérimentalement sur notre prototype).

Une intrusion au niveau de l'OS se caractérise par des flux d'information illégaux entre des processus et des fichiers, des sockets, des pipes, etc. Dans un graphe de flux d'information, une intrusion sera responsable de la création ou de la modification de flux d'information, d'objets actifs et/ou d'objets passifs. Une intrusion contre la confidentialité sera caractérisée par la création de flux d'information d'objets passifs vers des objets actifs et, si nécessaire, par la création de nouveaux objets. Une intrusion contre l'intégrité sera caractérisée par la création ou la modification de flux d'informations d'objets actifs vers des objets passifs et, si nécessaire, la création de nouveaux objets. Ainsi, une intrusion va affecter la valeur de la similarité entre les graphes de flux d'information d'un serveur compromis et d'un serveur non-compromis.

Seuil de similarité Le calcul de la similarité est une fonction prenant en paramètres deux graphes. Une valeur élevée de la similarité signifie que les serveurs se sont comportés d'une manière identique en ce qui concerne les flux d'information. Une valeur peu élevée de la similarité signifie que les serveurs se sont comportés de manière relativement différente, ce qui peut être la conséquence d'une intrusion ou d'une différence de conception ou de spécification. Il est nécessaire de déterminer un seuil en-dessous duquel une alerte est levée. Pour cela, nous avons décidé de procéder par apprentissage dans un contexte normal d'utilisation.

Le seuil doit être déterminé expérimentalement pour chaque couple de services diversifiés : la similarité dépend du cardinal des descripteurs des graphes considérés et donc dépend de l'application surveillée. Si les graphes sont grands (c'est-à-dire qu'ils ont beaucoup de nœuds et d'arcs), un ou plusieurs objets ou flux d'information non mappés auront moins d'influence sur le calcul de la similarité que si les graphes sont petits (c'est-à-dire qu'ils ont peu de nœuds et d'arcs). Ainsi, il n'est pas possible d'avoir un seuil unique pour toutes les applications.

Pour calculer ce seuil, il faut déterminer statistiquement quelle valeur correspond à un comportement normal, c'est-à-dire à une requête qui n'est pas une attaque. Ce seuil est forcément inférieur à 1 à cause des différences de conception. Au-dessus de ce seuil, il y a une forte probabilité que la différence détectée soit dûe à une différence de conception. En-dessous de ce seuil, la probabilité de détecter une différence de spécification doit être faible. Ce dernier cas produira une alerte, même si la différence entre les deux graphes n'est pas due à une intrusion : dans ce cas, cette alerte sera donc un faux positif.

Algorithme de détection d'intrusions Dans le contexte d'une architecture à $n = 3$ serveurs COTS S_i, il faut déterminer un seuil $t_{i,j}$ pour chaque couple de serveurs comme expliqué dans le paragraphe précédent. $t_{i,j}$ est le seuil choisi pour les serveurs S_i et S_j.

Détecter une intrusion requiert alors de calculer la similarité entre les graphes de chaque couple de serveurs pour une requête au service surveillé, ce qui correspond au calcul de $\frac{n\times(n-1)}{2}$ similarités entre graphes. Notons $s_{i,j}$ la similarité entre le graphe du serveur S_i et celui du serveur S_j. Comme la similarité est symétrique, nous avons $s_{i,j} = s_{j,i}$ pour tout (i,j) dans $\{1, n^2\}$. Notons $I_1^{i,j} = [0, t_{i,j}]$ et $I_2^{i,j} = [t_{i,j}, 1]$.

Nous avons choisi les règles suivantes pour déterminer la décision de l'IDS dans le cas de n serveurs :

$$\exists (i,j) \in \{1,n\}^2, i < j, s_{i,j} \in I_1^{i,j} \Rightarrow Alerte$$
$$\forall (i,j) \in \{1,n\}^2, i < j, s_{i,j} \in I_2^{i,j} \Rightarrow Pas\ d'alerte$$

c'est-à-dire qu'une alerte est émise dès qu'une similarité $s_{i,j}$ est faible (en-dessous de $t_{i,j}$). Aucune alerte n'est émise seulement lorsque toutes les similarités calculées sont supérieures à leur seuil respectif.

Localisation du serveur compromis La localisation du serveur compromis se base sur la règle suivante :

$$\left.\begin{array}{l} \exists i \in \{1,n\}, \forall j \in \{1,n\}, j \neq i, s_{i,j} \in I_1^{i,j} \\ \wedge \\ \forall (k,l) \in \{1,n\}^2,\ k \neq i, l \neq i, s_{k,l} \in I_2^{k,l} \end{array}\right\} \Rightarrow S_i \text{ est compromis}$$

Cette règle signifie que, sous l'hypothèse de décorélation des fautes, il ne doit y avoir qu'un seul serveur pour lequel les similarités entre ce serveur et les autres sont faibles, toutes les autres similarités calculées entre les autres serveurs devant être élevées. Dans tous les autres cas où l'IDS peut lever une alerte, la localisation du serveur compromis n'est pas possible.

	$s_{2,3} \in I_1^{2,3}$		$s_{2,3} \in I_2^{2,3}$	
$s_{1,2} \in$ \ $s_{1,3} \in$	$I_1^{1,3}$	$I_2^{1,3}$	$I_1^{1,3}$	$I_2^{1,3}$
$I_1^{1,2}$	$A/?$	A/S_2	A/S_1	$A/?$
$I_2^{1,2}$	A/S_3	$A/?$	$A/?$	NA

TABLE 4.1 – Alertes et localisation du serveur compromis dans le cas $N = 3$; A signifie alerte, NA signifie pas d'alerte, ? signifie que la localisation n'est pas possible, S_i signifie que le serveur en question est considéré comme compromis

Exemple pour trois serveurs Le tableau 4.1 résume, dans le cas de trois serveurs S_1, S_2 et S_3, dans quels cas une alerte doit être levée et s'il est possible de localiser le serveur compromis en fonction des similarités calculées.

La localisation n'est pas possible dans certains cas : par exemple, si $s_{1,2}$ et $s_{2,3}$ sont élevées et $s_{1,3}$ faible. Cela signifie que les comportements de S_1 et S_2 sont proches,

ainsi que les comportements de S_2 et S_3 mais que les comportements de S_1 et S_3 sont différents. Il n'est pas possible d'exhiber à partir du calcul de similarité un serveur dont le comportement est différent des deux autres.

4.3 Prototype et résultats expérimentaux

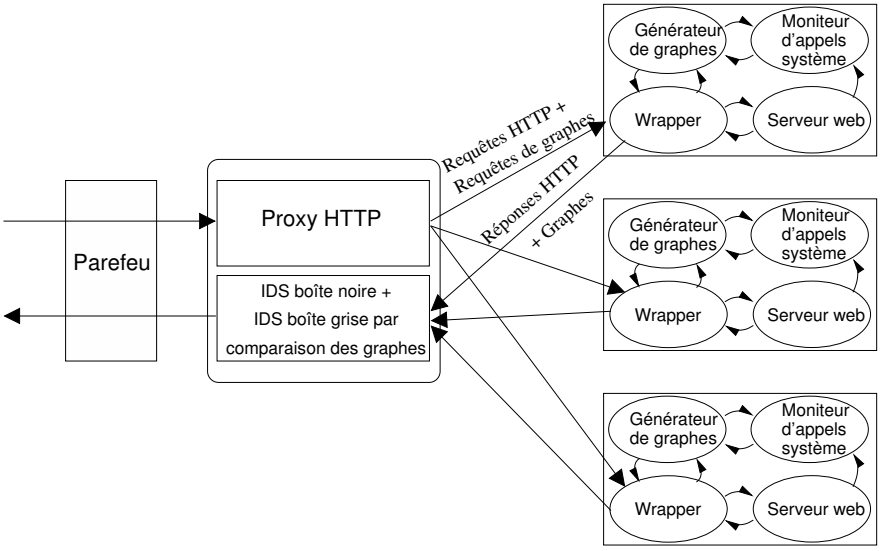


FIGURE 4.4 – Architecture et prototype

La figure 4.4 montre l'architecture qui a été mise en place pour valider l'approche proposée. L'architecture, comme dans le paragraphe précédent repose sur l'existence d'un proxy, d'un IDS et de trois serveurs diversifiés. La différence avec le prototype en boite noire réside dans l'information qui est traitée par l'IDS. Pour chaque requête, chaque serveur construit un graphe de flux d'information correspondant au traitement de la requête par le serveur web local. Afin de construire ce graphe localement, les appels systèmes générant des flux d'information sont interceptés sur chaque système d'exploitation (Windows, MacOS-X et Linux). Les graphes sont envoyés à l'IDS en plus de la réponse à la requête, qui calcule deux à deux leur similarité.

Corrélation entre l'approche boîte noire et l'approche boîte grise Dans cette architecture, nous utilisons les deux approches (boîte noire et boîte grise) conjointement. Si un des deux IDS lève une alerte, nous avons décidé que la combinaison des deux IDS lèvera également une alerte. Si aucune des deux méthodes ne lève une alerte, aucune alerte n'est levée.

Détermination des seuils $t_{i,j}$ Les seuils $t_{i,j}$ doivent être choisis expérimentalement pour chaque couple de serveurs utilisés. Nous avons décidé de les fixer de telle manière

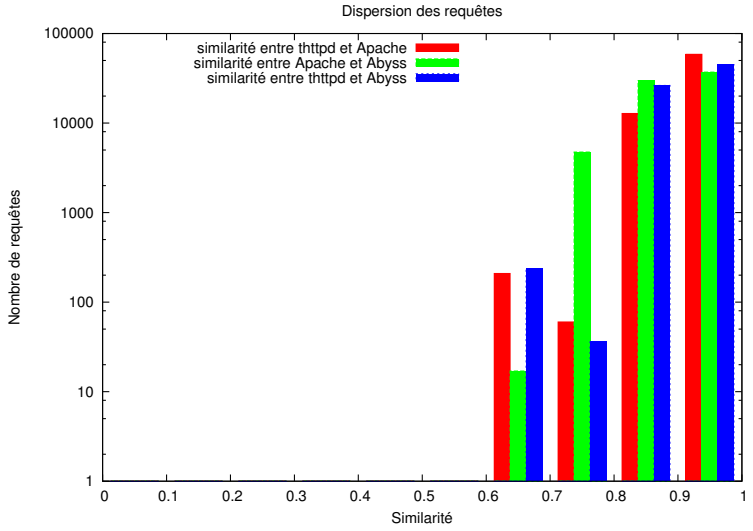


FIGURE 4.5 – Répartition des requêtes en fonction de la similarité pour chaque couple de serveurs (le nombre de requêtes est exprimé en échelle logarithmique)

que, pour la plupart des requêtes HTTP, notre prototype ne lève pas d'alertes, c'est-à-dire que la majorité des similarités calculées est supérieure aux seuils $t_{i,j}$.

Nous avons utilisé en tant qu'ensemble de requêtes normales, les requêtes provenant du fichier de log du serveur web du campus de Rennes de Supélec. Cet ensemble de requêtes couvre une semaine et est constitué de 71 596 requêtes. Pour vérifier que cet ensemble ne contient pas d'intrusion contre l'un des serveurs utilisés, nous avons utilisé l'IDS boîte noire présenté dans le chapitre précédent et WebSTAT [VRKK03]. Notre précédent IDS lève 197 alertes qui se sont révélées être des faux positifs. WebSTAT lève 33 alertes qui se sont également révélées être des faux positifs.

La figure 4.5 représente la répartition des requêtes pour les trois couples de serveurs utilisés lors de nos tests. Cette figure nous permet de déterminer les seuils $t_{i,j}$. Nous avons choisi de fixer les trois seuils à 0,7. Dans ce cas, plus de 99,5% des requêtes sont considérées comme normales par notre IDS (le nombre de requêtes sur la figure est exprimé en échelle logarithmique).

Notons S_1 le serveur thttpd, S_2 le serveur Apache et S_3 le serveur Abyss et $s_{i,j}$ les similarités correspondantes. Le tableau 4.2 résume les alertes levées par notre prototype pour la semaine d'apprentissage, tous les seuils étant fixés à 0,7. Notre prototype lève 247 alertes, ce qui représente un taux de faux positifs de 0,35%. L'opérateur de sécurité doit alors analyser environ 35 alertes par jour.

Résultats Pour évaluer l'IDS proposé, nous avons utilisé un autre ensemble de requêtes HTTP loggées par le serveur web de notre campus pendant une semaine. Cet ensemble est composé de 105 228 requêtes. WebSTAT génère seulement 4 alertes pour cette semaine : 3 alertes correspondent à des attaques qui échouent contre les serveurs de notre

	$s_{2,3} \in I_1^{2,3}$		$s_{2,3} \in I_2^{2,3}$	
$s_{1,3} \in$ / $s_{1,2} \in$	$I_1^{1,3}$	$I_2^{1,3}$	$I_1^{1,3}$	$I_2^{1,3}$
$I_1^{1,2}$	0 $(A/?)$	0 (A/S_2)	212 (A/S_1)	1 $(A/?)$
$I_2^{1,2}$	7 (A/S_3)	11 $(A/?)$	16 $(A/?)$	71 349 (NA)

TABLE 4.2 – Nombre d'alertes et identification du serveur considéré comme compromis pour la semaine d'apprentissage ; A signifie alerte, NA signifie pas d'alerte, ? signifie que la localisation n'est pas possible, S_i signifie que le serveur S_i est considéré comme compromis

architecture et la dernière ne correspond pas à une attaque et est donc un faux positif. Pour cette semaine de test, l'IDS boîte noire lève 50 alertes et l'IDS boîte grise 140 alertes (voir tableau 4.4). Le tableau 4.3 utilise la même notation que le paragraphe précédent et montre la localisation du serveur compromis lorsque c'est possible. Toutes ces alertes sont des faux positifs : elles ne sont en effet pas dûes à des intrusions. Cela représente un taux de faux positifs de 0,13% et 20 alertes par jour.

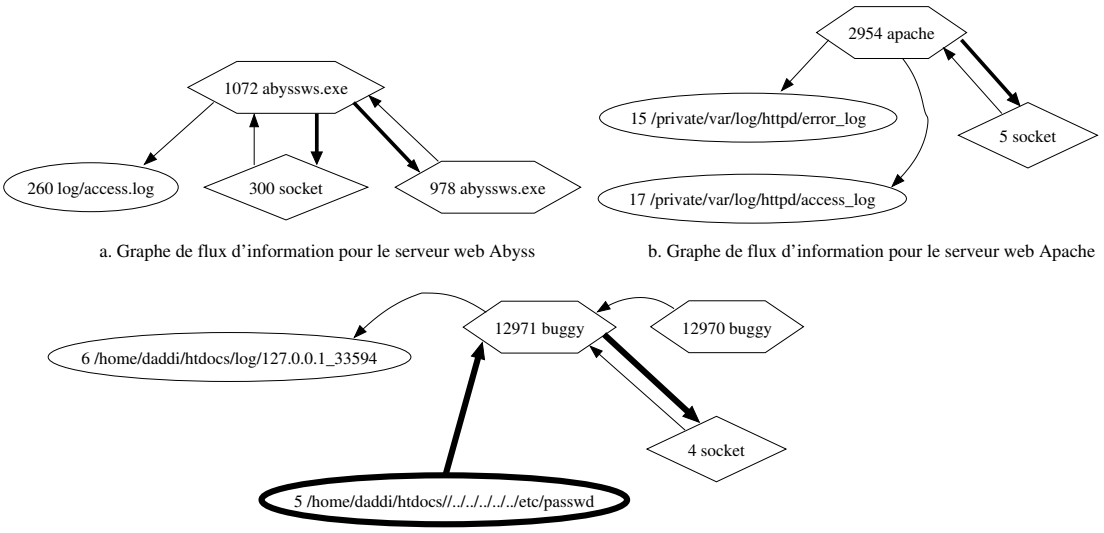
La combinaison des deux IDS lève 198 alertes : les deux IDS ne sont d'accord que sur deux requêtes. Cela représente un taux de faux positifs de 0,18% et environ 28 alertes par jour.

	$s_{2,3} \in I_1^{2,3}$		$s_{2,3} \in I_2^{2,3}$	
$s_{1,3} \in$ / $s_{1,2} \in$	$I_1^{1,3}$	$I_2^{1,3}$	$I_1^{1,3}$	$I_2^{1,3}$
$I_1^{1,2}$	0 $(A/?)$	0 (A/S_2)	120 (A/S_1)	1 $(A/?)$
$I_2^{1,2}$	2 (A/S_3)	1 $(A/?)$	16 $(A/?)$	105 088 (NA)

TABLE 4.3 – Nombre d'alertes et identification du serveur considéré comme compromis pour la semaine de test ; A signifie alerte, NA signifie pas d'alerte, ? signifie que la localisation n'est pas possible, S_i signifie que le serveur S_i est considéré comme compromis

IDS boîte grise / IDS boîte noire	alerte	pas d'alerte
alerte	2 (A)	48 (A)
pas d'alerte	138 (A)	105 040 (NA)

TABLE 4.4 – Corrélation entre les deux approches sur la semaine de test



a. Graphe de flux d'information pour le serveur web Abyss

b. Graphe de flux d'information pour le serveur web Apache

c. Graphe de flux d'information pour le serveur web vulnérable

FIGURE 4.6 – Diagnostic d'intrusions

4.4 Diagnostic d'intrusions

Afin de mettre en évidence les capacités de diagnostic de notre approche, nous nous mettons dans une situation où nous avons les trois serveurs web Abyss (sous Windows), Apache (sous MacOS-X) et buggyHTTP (sous Linux). Le serveur buggyHTTP est attaqué (attaque de type *directory traversal*), et le but de l'attaque est de récupérer le fichier */etc/passwd*. L'attaque réussit, et on obtient les trois graphes correspondant aux trois exécutions de la requête par les serveurs web. La figure 4.6 présente ces trois graphes. L'algorithme de détection trouve le meilleur *mapping* possible entre ces trois graphes pour maximiser les similarités. En gras, on trouve les éléments non *mappés* pour cette similarité maximale, et on s'aperçoit qu'ils présentent exactement le résultat de l'attaque, à savoir la lecture par le serveur buggyHTTP du fichier */etc/passwd*.

Cette propriété est extrêmement intéressante, car peu, voire aucun système de détection d'intrusion, n'est capable de fournir un diagnostic des alertes qu'il émet. Proposer à un administrateur de sécurité les trois graphes et mettre en évidence les éléments anormaux dans ces graphes lui permet d'avoir une vision extrêmement précise des intrusions ayant eu lieu au sein du système.

4.5 Travaux connexes

Les seuls travaux à notre connaissance qui s'approchent de notre contribution sont ceux décrits dans [GRS05]. Les auteurs mettent en place une distance comportemen-

tale entre des services diversifiés. Cette distance utilise la notion de séquence d'appels systèmes. Les auteurs utilisent la notion de *distance évolutionnaire* qui est utilisée pour calculer la distance entre des séquences ADN, et l'adapte à leur problème.

De même dans [GRS08], les auteurs utilisent une autre approche pour calculer la distance comportementale entre deux variantes : les chaînes de Markov cachées. L'approche est testée dans le cas de serveurs web et de serveurs de jeux.

4.6 Conclusion

Ce chapitre clôt les travaux que nous avons réalisés dans le cadre de la diversification fonctionnelle. Deux approches ont été proposées : une approche en boite noire, où seules les sorties réseau des variantes sont comparées, et une approche en boite grise où les graphes de flux d'information produits par l'exécution des variantes sont comparés. Bien que cette deuxième approche nous semble nettement moins efficace en terme de détection, elle apporte à la première approche des capacités de diagnostic qui sont très intéressants du point de vue d'un administrateur système. L'utilisation conjointe des deux approches peut donc permettre d'avoir une solution complète de détection et de diagnostic des attaques menées contre le système.

Afin d'être capable de diagnostiquer encore plus précisément quelle variante est compromise, il serait nécessaire d'avoir une idée du comportement interne de chaque application diversifiée. Ceci nécessite de mettre en place des mécanismes de détection au sein de l'application, par exemple en utilisant des contrôles de vraisemblance dans le logiciel. Les deux prochains chapitres de ce document vont justement porter sur ces types d'approches qui reposent sur la découverte de propriétés au sein du logiciel, et leur vérification à l'exécution.

Chapitre 5

Détection d'intrusion au sein du logiciel par invariants calculés statiquement

Dans tous nos travaux, nous supposons que la cible d'une attaque est une application. Dans les chapitres 3 et 4, nous proposons des approches architecturales permettant la détection de ces attaques contre l'application. Toutefois, ces solutions détectent l'intrusion une fois qu'elle s'est propagée en dehors de l'application attaquée. Il serait donc intéressant d'avoir des mécanismes de détection à l'intérieur de l'application afin d'être capable d'empêcher la propagation d'erreur à l'extérieur de l'application.

Comme dans toute approche de détection comportementale, nous devons construire un modèle de comportement du fonctionnement du logiciel qui puisse être vérifié à l'exécution. Comme nous l'avons vu dans l'état de l'art sur la détection d'intrusion au niveau applicatif, les modèles de comportement définissent généralement des propriétés sur le flot d'exécution d'une application, ou sur son flot de données. Alors que les méthodes qui se fondent sur le flot d'exécution sont matures et utilisables, les méthodes qui considèrent le flot de données sont extrêmement coûteuses en temps d'exécution. Parmi les méthodes de détection d'erreur utilisées en sûreté de fonctionnement, celles basées sur les contrôles de vraisemblance sont peu voire pas du tout utilisées pour la détection d'intrusion alors qu'elles induisent généralement de faibles surcoût à l'exécution.

Les travaux que nous présentons dans ce chapitre et le suivant portent justement sur ces contrôles de vraisemblance. Le but de ces contrôles est de vérifier qu'à certains points du programme, certaines valeurs de variables sont vraisemblables ou que des propriétés semblent valides. Le but de ces travaux est de générer automatiquement ces propriétés et leurs vérifications. Pour ce faire, nous devons découvrir des propriétés invariantes. Dans ce chapitre, ces invariants sont calculés de manière statique sur des programmes écrits en langage C. L'ensemble de ces travaux repose sur des fonctions

présentes dans l'environnement d'analyse statique *Frama-C* [CEA].

Ce chapitre s'architecture de la manière suivante : dans la section 5.1 nous présentons la problématique sur un exemple, puis dans la section 5.2 nous présentons le modèle de détection, et comment il est construit automatiquement. Enfin nous terminons par l'évaluation des mécanismes de détection en section 5.5.

Les travaux présentés dans ce chapitre ont été publiés dans IFIP SEC 2011 [DMTT11].

5.1 Exemple d'attaque et de violation d'invariant

Chen et al. [CXS+05] ont montré sur des cas réels que des attaques peuvent compromettre des données d'un programme sans pour autant altérer son flot de contrôle. En particulier, c'est le cas de la vulnérabilité de dépassement d'entier découverte en 2001 dans diverses implémentations de serveurs SSH, y compris l'une des plus répandues, à savoir OpenSSH. La donnée interne concernée par le dépassement est utilisée pour calculer un masque qui permet d'empêcher une opération d'écriture en dehors des limites d'un tableau. Un attaquant peut utiliser cette vulnérabilité pour modifier des données à n'importe quel endroit dans l'espace d'adressage du processus.

```
00: void do_authentication(){
01:   int authed = 0;
      ...
03:   if(!strcmp(passwd, ""))
        /* for users with no password */
05:   else
        /* do_authloop(); */
07:     while(authed != 1) {
08:       type = packet_read(data);
09:       switch (type) {
            ...
11:         case SSH_CMSG_AUTH_PASSWORD:

13:           authed=auth_password(passwd, data);
14:           break;
            ...
16:       }
17:     }
18:

19:   do_authenticated(user);
20: }
```

```
00: void do_authentication(){
01:   int authed = 0;
      ...
03:   if(!strcmp(passwd, ""))
        /* for users with no password */
05:   else
        /* do_authloop(); */
07:     while(authed != 1) {
08:       type = packet_read(data);
09:       switch (type) {
            ...
11:         case SSH_CMSG_AUTH_PASSWORD:
12:           assert(passwd[0] != '\0');
13:           authed=auth_password(passwd, data);
14:           break;
            ...
16:       }
17:     }
18:   assert(authed==0 ||
      (authed==1 && type==SSH_CMSG_AUTH_PASSWORD));
19:   do_authenticated(user);
20: }
```

FIGURE 5.1 – Code inspiré du serveur OpenSSH

Le code sur la partie gauche de la figure 5.1 est issu de celui du serveur OpenSSH. La vulnérabilité se trouve dans la fonction *packet_read*. On peut exploiter cette vulnérabilité pour s'authentifier sur le serveur SSH sans donner de login et mot de passe valide. Pour

cela, au moins deux types d'attaques peuvent être réalisées. La première consiste à modifier la valeur de la variable *authed*, afin de sortir de la boucle d'authentification sans fournir de mot de passe. La deuxième consiste à forcer le mot de passe (variable *passwd*) à une chaîne de caractère vide, ce qui permet à la fonction *auth_password* de renvoyer une authentification correcte. Bien sûr dans ces deux cas précis, l'attaquant peut être authentifié sous n'importe quel login, et en particulier en tant qu'administrateur.

Si on observe attentivement le programme, on s'aperçoit qu'il est possible de déterminer des propriétés normalement invariantes à certains points du programme. Ainsi, lorsqu'on arrive à la ligne 13 du programme, on sait que le mot de passe ne doit pas être vide (grâce au test ligne 3). Par conséquent, l'appel de la fonction *auth_password* ligne 13 avec comme argument un mot de passe de taille non nulle est un comportement normal invariant du programme. Si l'on observe la ligne 19, on s'aperçoit qu'il y a deux chemins distincts dans le programme qui permettent d'atteindre cette ligne. Le premier concerne un compte sans mot de passe : sur ce chemin la variable *authed* doit toujours garder une valeur nulle (initialisée ligne 01), puisqu'aucune authentification n'est nécessaire. Sur le deuxième chemin, on est sensé réaliser avec succès une authentification, ce qui signifie que le dernier type de données reçues par *packet_read* doivent être de type SSH_CMSG_AUTH_PASSWORD, et que la variable *authed* passe à la valeur 1. On a donc à la ligne 19 un invariant particulier qui est la disjonction de ces deux propriétés.

Le programmeur devrait, de lui-même, vérifier ces deux propriétés, pour la première ligne 12, et pour la deuxième ligne 18. Ces vérifications sont insérées dans la partie droite de la figure 5.1, et sont capables de détecter les attaques que nous avons décrites. Mais dans la pratique ce genre de vérification est malheureusement rarement effectuée, soit parce que le programmeur ne le fait pas, soit parce qu'il est difficile de trouver quelles sont les propriétés à vérifier. Le but de l'approche que nous proposons ici est de générer automatiquement ces vérifications, ce qui fournit une aide au programmeur.

5.2 Détection d'intrusion

Dans l'exemple ci-dessus, on s'aperçoit qu'en fait on calcule statiquement un comportement normal de l'application. Ce comportement est ensuite vérifié à l'exécution, afin de détecter d'éventuelles attaques. Toutefois, on note également que l'on ne capture qu'une partie des comportements de l'application : les comportements invariants. On définira dans notre approche un invariant comme une propriété qui est toujours vraie à un point précis du programme. Dans la pratique, les invariants ne sont pas suffisants pour décrire tous les comportements du programme, ce qui signifie que l'on s'attend à avoir des faux négatifs. Dans les sections suivantes, nous définissons notre modèle de comportement pour la détection d'intrusion, et nous expliquons comment le construire automatiquement. Enfin, nous montrons comment ce comportement peut être vérifié à l'exécution.

5.2.1 Modèle de détection orienté autour des données

Dans notre approche, nous considérons qu'un attaquant cherche à modifier des données dans l'espace d'adressage du processus afin d'exécuter un ou plusieurs appels systèmes illégaux. Un appel système est considéré comme illégal dans notre approche s'il est effectué dans un contexte erroné du programme (par exemple certaines variables ont des valeurs erronées). Exécuter ces appels systèmes illégaux peut être réalisé de deux façons différentes : soit l'attaquant altère des variables qui influencent le flot de contrôle interne du programme (par exemple des variables utilisées dans des conditionnelles) pour exécuter des appels dans un ordre incorrect, soit il modifie directement ou indirectement les valeurs d'un ou plusieurs paramètres utilisés dans ces appels systèmes (il exécute alors des appels systèmes dans un ordre correct, mais avec des valeurs erronées des paramètres). Ces deux types d'attaques modifient des données internes au programme, sans modifier des variables externes du système qui contrôlent le flot d'exécution de l'application (comme par exemple des adresses de retour dans la pile). Afin de détecter les modifications induites par une attaque, nous proposons d'identifier des contraintes invariantes sur les variables internes du programme qui doivent être vérifiées à l'exécution.

Dans notre approche, nous définissons le modèle de comportement associé à un appel système SC_i du programme le triplet (SC_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 doivent vérifier les variables de V_i. Les contraintes que nous utilisons dans notre approche sont des invariants, c'est-à-dire des propriétés qui sont toujours vraies à un point du code source du programme. Le modèle de comportement du programme se définit alors par l'union pour tous les appels systèmes réalisés par le programme de tous leurs comportements locaux : $MC = \{\forall i, (SC_i, V_i, C_i)\}$.

5.2.2 Découverte d'invariants

Afin de découvrir les invariants, il faut déterminer deux choses : d'une part les variables sur lesquelles ces invariants portent et, d'autre part, les types des invariants que l'on recherche.

Construction de l'ensemble des variables L'ensemble des variables qui nous intéressent pour un appel système donné est celui qui comprend toutes les variables qui ont une influence sur cet appel. Ainsi, on peut distinguer deux types de variables : les variables qui ont influencé le chemin d'exécution pour mener à l'appel système, ou les variables qui ont une influence sur les paramètres des appels système du programme. En analyse statique, on est capable, à partir d'un point d'intérêt du programme (ici un appel système), de calculer le sous-ensemble du programme qui a une influence sur ce point d'intérêt : cette technique est appelée le *slicing* [Wei82]. Dans notre approche, toutes les variables qui nous intéressent se trouvent dans le *slice* du programme ayant pour point d'intérêt un appel système. En analyse statique, le calcul du *slice* est

généralement fondé sur le calcul d'un graphe de dépendance du programme (*Program Dependency Graph ou PDG*) [KKP+81]. Le *PDG* est un graphe orienté dont les nœuds correspondent aux instructions et prédicats de contrôle, et les arêtes correspondent aux dépendances en données ou en contrôle. Ce graphe peut être utilisé pour mettre en évidence l'ensemble des variables qui influencent un appel système, et le type de dépendance. Dans notre implémentation, nous utilisons directement la notion de *PDG* pour calculer l'ensemble des variables qui influencent un appel système.

Découverte de contraintes invariantes Le calcul de contraintes vérifiées par le code source d'un programme nécessite l'utilisation de techniques d'analyse statique. En théorie, nous pourrions imaginer n'importe quel type de contraintes, comme par exemple des contraintes temporelles. En pratique, les techniques d'analyse statique calculent des contraintes invariantes, et toute technique capable de calculer des invariants peut nous être utile.

Parmi les techniques existantes, la méthode de calcul par interprétation abstraite [CC77] est l'une des plus usitées. Dans la pratique, l'interprétation abstraite fournit un moyen de calculer des propriétés sur des variables d'un programme en utilisant des domaines abstraits qui représentent des abstractions des propriétés réelles du programme. Plusieurs types de modèles ont été développés pour découvrir des invariants. Parmi ces techniques, nous nous sommes particulièrement intéressés à la construction de domaines abstraits numériques, c'est-à-dire que nous calculons des invariants sur les valeurs numériques des variables. Les domaines abstraits utilisables peuvent être classés en deux catégories : les domaines non relationnels qui calculent des propriétés sur les variables prises indépendamment les unes des autres, et les domaines relationnels qui permettent de calculer des propriétés sur des variables logiquement liées entre elles. Les domaines non relationnels incluent, par exemple : le domaine d'intervalles [CC77] qui permet de calculer des invariants de la forme $v_i \in [c_1, c_2]$ où v_i est une variable du programme et c_1 et c_2 des constantes numériques, le domaine de constantes ($v_i = c$) ou le domaine de congruence [Gra89] ($v_i \in a\mathbb{Z} + b$). Des exemples de domaines relationels peuvent être cités comme le domaine polyédrique [CH78] ($\alpha_1 v_1 + ... + \alpha_n v_n \leq c$), le domaine linéaire [Kar76] ($\alpha_1 v_1 + ... + \alpha_n v_n = c$) et le domaine linéaire avec congruence [Gra91] ($\alpha_1 v_1 + ... + \alpha_n v_n \equiv a[b]$). Le problème avec les domaines relationnels est qu'ils ne passent pas à l'échelle sur de gros programmes. C'est la raison pour laquelle *Frama-C*, sur lequel reposent nos travaux, utilise des domaines non relationnels.

Calcul d'invariants du *plugin* SIDAN dans Frama-C *Frama-C* fournit un *plugin* nommé *Value Analysis* qui implémente par interprétation abstraite le calcul du domaine de variation de variables numériques en un point donné du programme. Les invariants que nous générons reposent sur l'utilisation de ce *plugin* : nous construisons des invariants qui reposent sur l'évaluation des valeurs que peuvent prendre des variables numériques. Le *plugin* calcule, pour un point du programme et une variable donnée, son domaine de variation sous la forme d'un intervalle $[a, b]$.

```
00: extern int a, b;
01: void f(int);

03: void g(){
04:   if (b == 0) a = 1;
05:   else if(b == 1) a = 2;
06:   else return;



09:   f(a);
10: }
```

```
00: extern int a, b;
01: void f(int);

03: void g(){
04:   if (b == 0) a = 1;
05:   else if(b == 1) a = 2;
06:   else return;

08:   assert((a == 1 && b == 0) ||
              (a == 2 && b == 1))
09:   f(a);
10: }
```

FIGURE 5.2 – Exemple de code C qui met en valeur les relations entre variables

Si on considère le programme C de la figure 5.2, f étant l'appel système, nous allons essayer de construire un invariant pour l'appel ligne 9 en utilisant le domaine de variation des variables dont il dépend, à savoir les variables a et b. Si on utilise le *plugin Value Analysis* de *Frama-C*, on obtient que ligne 9, $a \in \{1, 2\}$ et $b \in \{0, 1\}$. Le domaine abstrait utilisé par le *plugin Value Analysis* étant non relationnel, il perd les relations entre les variables a et b. Or, si on analyse le programme à la ligne 9, on s'aperçoit que quand la variable b est égale à 0 alors a est égale à 1, et quand la variable b est égale à 1 alors a est égale à 2. En fait, pour obtenir ce résultat, il faut considérer qu'il existe deux chemins qui mènent à la ligne 9, et qu'on peut calculer un invariant sur a et b pour chacun de ces chemins pris séparément. Par défaut, le *plugin Value Analysis* fait une union des domaines de valeurs d'une même variable lorsque deux chemins se rejoignent. Par contre, en interne, il est capable de garder en mémoire les différents domaines possibles pris par chaque variable pour chacun des chemins. Dans la pratique, on est capable de le paramétrer pour définir le nombre de chemins explorés en parallèle, ce qui va définir le nombre de domaines de valeurs qu'il va garder en mémoire avant de réaliser une union des domaines calculés sur les différents chemins.

Le *plugin SIDAN* que nous avons développé utilise cette capacité du *plugin Value Analysis*. Toutefois, par défaut les états internes calculés en parallèle ne sont pas directement accessibles. Il faut pour cela "intercepter" (utilisation d'un *hook*) le fonctionnement du *plugin Value Analysis* afin d'accéder à son état interne. Pour l'exemple présenté sur la figure 5.2, on obtient (ligne 9) que sur l'un des 2 chemins menant à l'appel de f, $(b == 0 \wedge a == 1)$ et sur l'autre chemin, $(b == 1 \wedge a == 2)$. L'un au moins de ces deux invariants devant être vrai, la disjonction des deux propriétés doit être vraie, à savoir $(((b == 0) \wedge (a == 1)) \vee ((b == 1) \wedge (a == 2)))$. Les invariants que l'on construit sont donc des disjonctions des propriétés calculées sur les différents chemins menant à l'appel système f.

Il faut noter ici que les invariants que nous calculons portent sur des entiers. En pratique, le *plugin Value Analysis* est capable de calculer des domaines de variation sur des entiers et des flottants. Par contre, il n'est pas vraiment efficace sur des pointeurs, pour lesquels il détecte au mieux qu'ils pointent sur une zone de mémoire non allouée, ou en dehors de bornes d'une zone mémoire. De plus, il est impossible de faire du calcul de valeur sur des pointeurs alloués dynamiquement, ni sur des chaînes de caractères, car les chaînes de caractères en C se comportent comme des tableaux.

5.3 Génération d'assertions exécutables

Une fois les invariants calculés, il faut être capable de les vérifier à l'exécution. Afin d'effectuer cette vérification, nous insérons dans le code des assertions exécutables qui seront compilées, puis exécutées. Ce type d'approche peut être assimilé à de la programmation défensive. Toutefois, pour des raisons que nous allons décrire, nous n'allons pas uniquement générer des assertions qui concernent des appels systèmes, mais également tous les autres appels de fonctions. En effet :

1. Lorsque nous calculons un invariant pour un appel système, les variables qui sont accessibles par l'assertion exécutable sont les variables locales à la fonction dans laquelle est réalisé l'appel. Nous ne pouvons donc pas vérifier des invariants portant sur des variables qui sont dans la pile des appels qui ont mené à cette fonction. Afin de réaliser cette vérification, il est donc nécessaire de distribuer des vérifications sur les chemins qui peuvent mener à cette fonction. Nous avons choisi de calculer des invariants pour tous les appels de fonction sur les chemins qui mènent à l'appel système.
2. Lorsque nous rencontrons un appel de fonction, la question se pose de savoir s'il s'agit d'un appel système ou non. Il n'est pas aisé de répondre à cette question de manière automatique. En effet, l'appel peut se faire vers une librairie externe dont on ne connait pas les sources, et qui peut contenir un appel système sans qu'on le sache.

Il y a une solution simple pour résoudre ces deux problèmes : calculer des invariants, et vérifier les assertions exécutables correspondantes, pour tous les appels de fonction que l'on rencontre dans le programme. C'est cette solution qui a été retenue dans le cadre du *plugin* SIDAN. Ce choix a un impact sur le modèle de comportement original. Le modèle de comportement normal pour un appel de fonction F_i est (F_i, V_i, C_i) où V_i est l'ensemble des variables dont dépend l'appel de fonction, et C_i les contraintes invariantes que l'on calcule sur ces variables. Le modèle de comportement normal devient $MC_2 = \{\forall i, (F_i, V_i, C_i)\}$. On remarque que le modèle de comportement normal original $MC \subset MC_2$, de sorte que le nouveau modèle de comportement définit plus de contraintes. En ce sens, le modèle MC_2, qui est une sur-approximation de MC, peut lever plus d'alertes que le modèle original, puisqu'il contient plus de contraintes. Ainsi, le modèle pratique mis en place peut donc lever de fausses alertes, si on considère qu'une alerte ne correspondant pas à un appel système incorrect, mais seulement à un

appel de fonction incorrect, est une fausse alerte.

Afin de démontrer la capacité de l'approche à générer des invariants dans des cas réels, nous avons mené différentes expérimentations. La première expérimentation a été de générer des invariants sur OpenSSH, dont une fraction du code est donnée en exemple sur la figure 5.1. Pour l'attaque que nous avons décrite, les invariants donnés sur cette figure ont été découverts, et les assertions générées.

La deuxième série d'expérimentations a consisté en l'instrumentation de programmes de la suite SPEC 2006. Cette suite propose de mesurer la performance d'un CPU en utilisant un ensemble de programmes. Nous avons fait le choix de choisir cette suite de programmes, car elle va nous permettre de mesurer de manière précise la surcharge à l'exécution de mécanismes de détection générés (cf. section 5.4). Le tableau 5.1 contient le résultat de cette instrumentation. Nous pouvons constater que pour tous les programmes, nous avons réussi à découvrir des invariants, et généré des assertions. Le pourcentage d'appels de fonctions pour lesquelles on est capable de générer une assertion exécutable est variable d'un programme à l'autre. Le pire des cas dans cette série de tests est celui de (*milc*) où 23% des appels de fonctions sont vérifiés, et le meilleur *bzip2* où 72% des appels de fonctions sont sujets à vérification. Cette variabilité est liée à plusieurs paramètres : par exemple, nous calculons des invariants uniquement pour des variables entières, et certains appels de fonctions dépendent de variables d'un autre type. Même pour des variables de type entier, il n'est pas toujours possible de calculer leur domaine de variation de manière statique, par exemple parce que ces variables dépendent d'entrées utilisateur à l'exécution, ou du contenu de fichiers de configuration, etc.

D'autre part, nous pouvons constater que le temps de calcul nécessaire à l'instrumentation est excessivement variable d'un programme à l'autre. Cette variabilité est difficilement explicable, puisqu'elle est liée au fonctionnement interne de *Frama-C* et de ses *plugins*.

	lbm	mcf	libquantum	bzip2	milc	sjeng
Nombre de lignes de code	1267	2077	3567	7292	12837	13291
Nombre d'appels de fonctions	72	88	228	134	1274	1718
Temps de calcul (mn)	12	8	122	1044	4123	5245
Nombre d'assertions	28	46	114	96	293	729
Taux d'instrumentation	38%	52%	50%	72%	23%	42%

TABLE 5.1 – Résultats de l'instrumentation d'applications locales

Une troisième série d'expérimentations a été menée sur des services réseaux divers, tels que des serveurs SSH, SMTP et HTTP. Le taux d'instrumentation des fonctions obtenu est relativement élevé, excepté dans le cas de fnord (un petit serveur web), où

seules 3% des fonctions donnent lieu à une vérification. La raison principale de ce faible score pour ce programme est lié à sa nature hautement dynamique, de sorte qu'il est difficile de calculer statiquement des invariants.

	dropbear	ssmtp	fnord	ihttpd	nullhttpd
Nombre de lignes de code	11177	2717	2622	1180	5968
Nombre d'appels de fonctions	429	314	232	289	399
Temps de calcul (mn)	962	19	224	1	317
Nombre d'assertions générées	257	237	6	109	234
Taux d'instrumentation	60%	75%	3%	38%	58%

TABLE 5.2 – Résultats de l'instrumentation de services réseau

5.4 Surcharge à l'exécution

Les programmes issus de SPEC 2006 étant destinés à faire des mesures de performances, il est judicieux de comparer les scores des différents programmes avant et après leur instrumentation par SIDAN. Ces mesures sont résumées dans le tableau 5.3. On constate que dans le pire des cas, la perte de score induite par les assertions exécutables générées est inférieure à 1%. Ce qui est très faible, et est dû au nombre peu élevé d'instructions qui doivent être exécutées en supplément du programme original après instrumentation.

	lbm	mcf	libquantum	bzip2	milc	sjeng
Score pour le binaire non instrumenté	19.24	17.80	23.62	16.59	12.64	18.41
Score pour le binaire instrumenté	19.17	17.72	23.45	16.54	12.59	18.24
Perte de score	0.37%	0.45%	0.72%	0.30%	0.40%	0.93%

TABLE 5.3 – Mesure de surcharge due à l'instrumentation

5.5 Evaluation des mécanismes de détection

À ce point du document, on sait qu'on est capable de générer automatiquement des assertions exécutables, même sur des programmes importants en terme de nombre de lignes, et que ces assertions coûtent peu en temps de calcul. Il convient maintenant d'évaluer leur capacité de détection sur des cas réels.

5.5.1 Simulation d'attaques

Afin d'évaluer le taux de faux négatifs, il est nécessaire de connaître toutes les attaques contre chaque logiciel instrumenté, et de vérifier si chacune de ces attaques

est bien détectée ou non. Avoir une telle connaissance est bien sûr irréalisable. C'est pourquoi nous nous sommes tournés vers une méthode d'évaluation de mécanismes de détection d'erreur : l'injection de fautes.

En sûreté de fonctionnement, l'injection de fautes est dictée par le modèle de faute que l'on considère. Par exemple, pour des fautes matérielles sur une mémoire, on peut prendre comme modèle de fautes la modification d'un bit altéré aléatoirement dans la mémoire. Ce type de modèle est relativement simple à reproduire en modifiant de manière logicielle la mémoire, afin de vérifier si les mécanismes de détection d'erreurs fonctionnent bien correctement.

Dans notre cas, le modèle de fautes est considérablement plus complexe. Une attaque n'est généralement pas aléatoire (elle utilise une vulnérabilité très précise, avec des entrées très précises), toutefois sa conséquence, elle, est très bien définie : il s'agit de modifier une ou plusieurs variables parmi un choix restreint en leur donnant des valeurs précises. Pour que l'attaque fonctionne, il faut connaître ces valeurs, ce qui est un problème particulièrement difficile. Nous avons donc décidé de mettre en place un modèle d'injection approximatif : il reproduit le résultat de l'attaque en modifiant une variable qui a un sens du point de vue de l'attaquant (elle permet d'influencer un appel de fonction), alors que la valeur affectée à cette variable sera, elle, aléatoire. Bien que cette méthode ne soit pas forcément idéale pour évaluer des mécanismes de détection d'intrusion (on n'est jamais certain que l'injection correspond à une attaque réelle réussie), elle permet de se donner une idée de la capacité de détection des mécanismes générés.

5.5.2 Instrumentation de code et injection de faute

```
extern int a;
const int b = 1;

if (a == 0) {
  inject(0,2,&a,sizeof(a),&b,sizeof(b));
  assert(b == 1 && a == 0);
  f(b);
}
```

$A \equiv I$

```
extern int a;
extern int b;

if (a) {
  inject(0,2,&a,sizeof(a),&b,sizeof(b));
  assert(a != 0);
  f(b);
}
```

$A \subset I$

```
extern int a;
extern int b;

if (a == b) {
  inject(0,2,&a,sizeof(a),&b,sizeof(b));

  f(b);
}
```

$A \equiv \emptyset$

FIGURE 5.3 – Exemples de mécanismes d'injection

Afin de déterminer le modèle d'injection de fautes, deux problèmes doivent être résolus : (1) comment déterminer l'ensemble des variables qui sont potentiellement les cibles d'une attaque et (2) comment déterminer à quel moment la valeur de chaque variable doit être perturbée. Il ne faut pas oublier qu'on veut perturber le contexte dont dépend un appel de fonction. Pour déterminer ce contexte, il faut, comme précédemment, calculer le *slice* ayant pour point d'intérêt l'appel de fonction. Afin de choisir le mo-

ment où une variable peut être perturbée, nous insérons du code dans l'application qui perturbe le contexte juste avant l'appel de fonction. Ce code se caractérise par l'appel à une fonction nommée *inject*, comme précisé sur la figure 5.3. Sur cette figure, on distingue deux types d'ensemble de variables : l'ensemble A des variables qui interviennent dans des assertions exécutables avant les appels de fonctions, et l'ensemble I des variables dans lesquelles on peut réaliser des injections. Ces deux ensembles sont tous les deux calculés à partir de l'ensemble des variables contenues dans le *slice* dont le point d'intérêt est l'appel de fonction. L'ensemble I est égal à cet ensemble. Toutefois, pour certaines variables il n'est pas possible de calculer statiquement un invariant par la méthode que l'on utilise dans notre approche. Par conséquent, l'ensemble A est toujours inclus dans l'ensemble I. Dans le pire des cas (cf. le code le plus à droite de la figure 5.3), on est capable de perturber un ensemble de variables par injection de faute sans qu'aucune assertion n'ait été produite faisant intervenir ces variables.

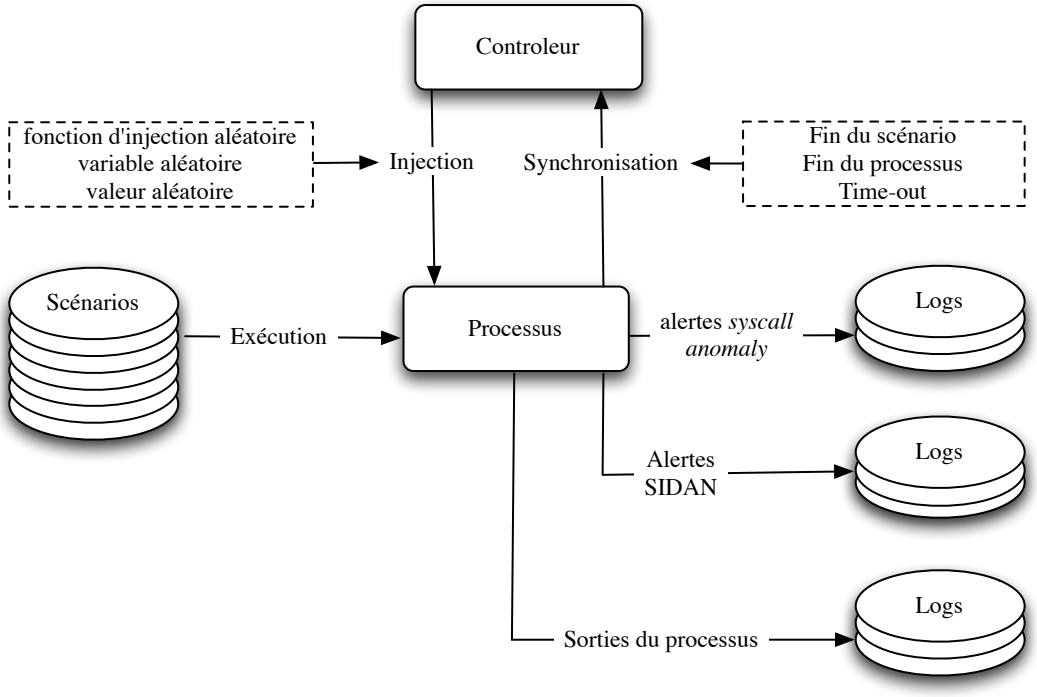


FIGURE 5.4 – Protocole expérimental

Chaque point d'injection dans le logiciel instrumenté se voit assigner un identificateur unique (ici 0 sur la figure 5.3), qui permet, depuis l'extérieur du processus de déterminer pour une exécution donnée lequel doit être activé. Les autres arguments de la fonction *inject* sont le nombre de variables qui peuvent être manipulées (2 variables sur la figure 5.3) et, pour chacune d'elles, son adresse en mémoire et sa taille. L'injection

est pilotée par un contrôleur qui, pour chaque exécution, détermine l'injection qui doit être réalisée. Une seule injection peut être réalisée durant une exécution particulière du programme. Pour un scénario d'exécution donné (c'est-à-dire pour une série déterminée d'entrées du programme), on choisit aléatoirement un point d'injection atteignable par cette exécution, ainsi qu'une variable et une valeur pour cette variable. Le processus est décrit sur la figure 5.4. La difficulté de l'approche consiste à déterminer des scénarios suffisamment complets pour s'assurer qu'on couvre le maximum d'appels de fonctions du code. Ce problème est clairement un problème de test logiciel.

5.5.3 Résultats expérimentaux

		SIDAN	*syscall-anomaly*	défaillance par arrêt
ihttpd	Injections détectées	6419	4383	1167
ihttpd	Taux de détection	11,34%	7,74%	2,06%
nullhttpd	Injections détectées	29107	21337	9184
nullhttpd	Taux de détection	30,03%	22,01%	9,47%

TABLE 5.4 – Résultats de détection

Afin d'évaluer la couverture de détection des assertions générées dans le logiciel, nous effectuons deux types de mesures : la première est le taux de détection de nos mécanismes, la deuxième est le taux obtenu avec un IDS nommé *syscall anomaly*. Cet IDS repose sur l'observation de séquences d'appels systèmes et de leurs arguments. Il s'agit sans aucun doute de l'IDS le plus à même de détecter le type d'attaques auxquelles nous nous intéressons. Il faut noter ici que ce dernier IDS a été configuré pour donner les meilleurs résultats possibles (un profil différent a en particulier été utilisé pour chaque scénario d'exécution).

Les expérimentations ont été menées sur deux serveurs HTTP préalablement instrumentés. Dans le cas de *ihttpd*, 56584 injections ont été réalisées, contre 96925 dans le cas de *nullhttpd*. Nous remarquons dans le tableau 5.4 que 11% et 30% des injections ont été détectées respectivement dans le cas de *ihhtpd* et *nullhttpd* par notre approche. Ces résultats peuvent sembler faibles, mais si on les compare à ceux de *syscall-anomaly* (respectivement 7% et 22%), on obtient des résultats sensiblement meilleurs dans notre approche, ce qui est intéressant compte-tenu du fait que, de surcroît, le surcoût induit par nos mécanismes est extrêmement faible. De plus, il ne faut pas oublier que *syscall anomaly* a été utilisé de manière optimale, avec un profil différent pour chaque exécution, ce qui n'est jamais le cas dans la réalité.

5.6 Conclusion

L'approche de détection par invariants proposée dans ces travaux montre qu'il est possible de construire de manière automatique un modèle de comportement d'un logiciel, en calculant statiquement des propriétés invariantes de ce logiciel. Toutefois, on note que les invariants proposés sont relativement pauvres, puisqu'ils ne portent que sur des domaines de variations de variables. Malheureusement, il est reconnu qu'utiliser des modèles abstraits relationnels (qui nous permettraient de calculer des invariants plus complexes) ne passe pas très bien à l'échelle. C'est la raison pour laquelle, nous avons abordé, dans le dernier chapitre, une méthode dynamique pour calculer des invariants.

Chapitre 6

Détection d'intrusion au sein du logiciel par invariants calculés dynamiquement

6.1 Introduction

Comme dans le chapitre précédent, les travaux présentés dans ce chapitre visent à détecter des attaques contre des applications qui sont en interaction avec des clients, la plupart du temps au travers du réseau.

Nombre de ces applications serveurs sont configurées par l'intermédiaire de fichiers de configuration, de bases de données, etc. C'est par exemple le cas de serveurs HTTP, FTP, ou d'applications web. Le recours à de telles méthodes est particulièrement pratique pour l'administrateur, puisqu'elles lui permettent de modifier à sa guise le comportement du logiciel qu'il déploie. Toutefois, cela a un impact non négligeable sur le code source du programme lui-même. En effet, un grand nombre des constantes du programme sont absentes de son code source, de sorte que le calcul automatique d'invariants par analyse statique présenté dans le chapitre précédent ne permet pas de capturer des propriétés invariantes sur les données du programme.

La notion d'invariants calculés dynamiquement propose une approche permettant de pallier ce problème. Afin de calculer dynamiquement des invariants sur les valeurs prises par des variables d'un programme, il est nécessaire d'observer un ensemble d'exécutions normales du programme, et de sauvegarder les valeurs des variables du programme durant ces exécutions. La conséquence d'une telle approche est que l'on a la connaissance des valeurs que prennent les variables à l'exécution, valeurs qui ne sont pas connues dans le code source du programme. On est alors capable de générer des invariants qui ne pourraient en aucun cas être calculés statiquement.

Dans ce chapitre, nous montrons dans la section 6.2 la limite de l'approche statique

sur un exemple, et les types d'invariants que nous espérons calculer dynamiquement, sur une application écrite en langage C, puis sur une application web interprétée. Nous définissons ensuite un modèle de comportement de l'application construit dynamiquement (section 6.3) et exposons brièvement deux implémentations (sections 6.4 et 6.5). Enfin, dans le cadre de l'une de ces implémentations, nous exposons une évaluation des mécanismes de détection générés automatiquement par notre approche section 6.6.

6.2 Invariants dynamiques dans une application

6.2.1 Limite de l'approche précédente

```
1. int main(int argc, char ** argv)
2. {
3.     char buffer[256];
4.     uid_t uid;
5.
6.
7.     uid = getuid(); /* récupération du propriétaire du processus */
8.
9.     while(gets[buffer])
10.    {
11.         seteuid(0); /* passage en privilège root */
12.         printf(buffer); /* instruction exécutée en mode privilégié */
13.         seteuid(uid);
14.    }
15.
16. }
```

FIGURE 6.1 – Vulnérabilité Format String

La figure 6.1 montre un exemple classique d'attaque utilisant la vulnérabilité de type *Format String*. L'utilisateur donne en entrée une chaîne de caractères (ligne 9) qui est utilisée directement ligne 12, sans que l'on définisse son format. En mettant certains caractères de format dans cette chaîne, l'attaquant est capable d'écrire dans une variable du programme de son choix, ici, la variable *uid* qui contient l'identificateur de l'utilisateur. Le résultat de cette manipulation est que l'attaquant est capable d'obtenir les droits administrateurs (ligne 13). Si on revient à la méthode de calcul statique d'invariants présentée dans le chapitre précédent, on cherche à obtenir un invariant sur la valeur de la variable *uid* utilisée lors de l'appel système à la fonction *seteuid*. Malheureusement, le calcul de valeur de *uid* ne mène à aucun résultat, la variable étant définie de manière externe ligne 7.

Pourtant, si on exécute ce programme dans un contexte libre d'attaques, on s'aperçoit de deux choses :

1. la variable *uid* reste constante entre le moment où on lui affecte sa valeur ligne 7 et la fin du programme ;
2. la variable garde la même valeur d'une itération de la boucle à l'autre.

Le calcul statique de valeur utilisé dans le chapitre précédent ne permet pas de calculer le premier type d'invariant (la valeur étant inconnue). De plus le deuxième définit une relation temporelle entre les évolutions successives d'une valeur d'une variable dans le temps, ce qui ne peut être facilement capturé que dynamiquement.

6.2.2 Exemple d'invariant dynamique dans une application web

```
1. class UsersController < ApplicationController
...
16.  def login
17.    if request.post?
18.      # whole user is stored in the session
19.      if session[:user] = User.authenticate(
                                      params[:user][:login],
                                      params[:user][:password])
20.        flash[:notice]  = 'You have been successfully logged in.'
21.        if session[:user].admin
22.          redirect_to :controller => '/admin/home', :action => 'index'
23.        else
24.          redirect_to :controller => '/user/home',  :action => 'index'
25.        end
26.      else
27.        flash.now[:error] = 'Login failed'
28.      end
29.    end
30.  end
  ...
51.end
```

FIGURE 6.2 – action login du contrôleur User

Dans les travaux que nous avons menés sur les invariants dynamiques, nous nous sommes particulièrement intéressés à des applications web qui sont tout à fait caractéristiques de ces applications qui utilisent peu de constantes dans le code source du programme. Non seulement ils dépendent généralement de données stockées dans une base de données, mais de plus chaque page web dynamique est en mesure de recevoir des paramètres dont on ne connaît pas forcément la valeur dans le code source

de la page. La figure 6.2 est un extrait du code d'une page écrite en Ruby ayant cette caractéristique. Le client web invoque la fonction de *login* de l'application web en lui passant en paramètre les *login* et mot de passe de l'utilisateur. Ces *login* et mot de passe, inconnus avant l'exécution, sont vérifiés en faisant appel aux informations contenues dans la base de données (appel à la fonction *User.authenticate*). En comportement normal, l'utilisateur retourné par la requête à la base de données possède un champ *login* qui est égal au *login* passé en paramètre par le client. On a donc un invariant qui exprime que $session[: user].login == params[: user][: login]$, et qui est impossible à calculer statiquement, puisque cela nécessiterait de prendre en compte le contenu de la base de données.

Parmi les attaques contre les données les plus classiques dans le cadre d'application web, on compte les injections SQL. Dans le cas présent, une injection SQL consisterait à donner en entrée du paramètre `login` la chaine de caractère "`'OR '1'='1`". Cette chaine est ensuite concaténée au reste de la chaine constituant la requête SQL, qui est ressemblante à un "`SELECT * FROM Users where login='$params[:user][:login]' OR '1'='1'`". La condition de sélection dans la base est toujours vraie, de sorte qu'on obtient tous les utilisateurs de la table *Users*. Le premier utilisateur de cette table est donc sélectionné, à savoir (dans notre cas) l'administrateur dont le champ *login* a la valeur *root*. Par conséquent, l'attaque réussit et viole l'invariant précédemment défini.

6.3 Modèle de comportement à base d'invariants dynamiques

Dans cette section, nous allons montrer comment nous pouvons construire un modèle de comportement fondé sur des invariants calculés dynamiquement. Trois types d'invariants sont découverts dans notre approche :

1. des relations entre des variables au même point d'exécution du programme ;
2. des relations entre les valeurs d'une même variable à différents points d'exécution ;
3. des relations entre différentes variables à des points d'exécution différents.

Afin de découvrir les relations entre les variables, nous utilisons un outil issu du test logiciel nommé Daikon [EPG+07]. Cet outil est capable, à partir de traces sur des valeurs de variables observées à l'exécution, d'inférer des invariants. Ces invariants sont les relations que l'on cherche à calculer.

Dans les paragraphes qui suivent, nous allons formaliser le modèle d'exécution, ce qui nous permet de définir l'ensemble des variables sur lesquelles nous allons calculer ces invariants.

Modèle d'exécution L'observation d'une exécution nécessite la définition de la notion de point d'exécution. Dans ces travaux, nous définissons la notion de point d'exécution par le triplet $p = (pc, mem, t)$ où pc est le compteur ordinal, mem est l'état

de la mémoire associé avec le programme exécuté, et t le temps logique au moment où le point d'exécution est atteint.

L'état de la mémoire mem associé à une exécution en un point d'exécution est l'ensemble des variables locales et globales qui sont accessibles à ce point d'exécution. Si on note σ une exécution du programme, nous définissons $S_{p,\sigma}$ l'état du programme pour une exécution σ au point d'exécution p. $S_{p,\sigma}$ est en fait une fonction depuis l'ensemble des variables vers l'ensemble des valeurs qu'elles peuvent prendre. Ainsi $S_{p,\sigma}(v)$ est la valeur que prend la variable v pour l'exécution σ au point d'exécution p.

Les invariants que nous désirons calculer doivent être valides pour toutes les exécutions σ du programme considéré. Alors qu'il est probablement impossible de définir toutes les exécutions possibles, nous définissons Σ un ensemble d'exécutions qui permettent d'avoir une idée représentative du comportement normal du programme.

Afin de définir sur quelles valeurs nous allons calculer les invariants, il est nécessaire de définir une relation d'équivalence entre deux points d'exécution d'un programme $p_1 = (pc_1, mem_1, t_1)$ et $p_2 = (pc_2, mem_2, t_2)$. On dit que $p_1 \in \sigma_1$ et $p_2 \in \sigma_2$ sont équivalents (noté $p_1 \equiv p_2$) si et seulement si leurs compteurs ordinaux sont égaux, c'est-à-dire si $pc_1 = pc_2$. Deux points d'exécution sont donc équivalents s'il correspondent à la même position dans le programme exécuté, sans considération de l'état de la mémoire et du moment de l'occurrence de ces points d'exécution. Cette relation d'équivalence est compatible avec la façon dont Daikon découvre des invariants. En effet, pour Daikon, deux points d'exécution sont équivalents tant qu'ils concernent le même compteur ordinal. Daikon considère en fait trois points d'exécution différents :

1. les points d'entrée de fonctions ;
2. les points de sortie de fonctions ;
3. les instructions dans le corps des fonctions.

Dans la littérature, Daikon est utilisé pour déterminer des invariants pour les deux premiers types de points d'exécution, bien qu'il soit capable d'en trouver pour le troisième type. Tirer profit de cette possibilité est l'une des contributions de ces travaux, et va nous permettre de découvrir des invariants à n'importe quel point d'exécution du programme.

Ensemble des variables Nous introduisons finalement la notion d'état unifié pour une variable v pour l'ensemble des exécutions Σ pour un compteur ordinal pc comme la fonction définie par :

$$S_{pc}(v) = \bigcup_{\sigma \in \Sigma} [\bigcup_{p \in \sigma | p = (pc, \star, \star)} S_{p,\sigma}(v)].$$

Cette fonction peut être expliquée comme suit : pour un compteur ordinal pc donné, un état unifié décrit l'ensemble des valeurs qui peuvent être accessibles à ce point du programme (c'est-à-dire pour tous les points d'exécution équivalents). De cette définition,

nous pouvons déduire l'ensemble de toutes les variables V_{pc} à un point d'exécution donné : $V_{pc} = \{v \mid S_{pc}(v) \neq \{\bot\}\}$. Cet ensemble contient toutes les variables qui peuvent être utilisées par Daikon pour calculer un invariant pour un compteur ordinal donné.

Deux points doivent être notés concernant ce modèle :
- Premièrement, il est nécessaire de sauvegarder toutes les variables chaque fois que le compteur ordinal évolue. Cette méthode a l'inconvénient de nécessiter une grande quantité de mémoire. Il est donc nécessaire de réduire cet ensemble à un ensemble plus réduit et utilisable.
- Deuxièmement, découvrir des invariants sur ces variables nécessite de définir quels types d'invariants sont intéressants pour détecter des attaques.

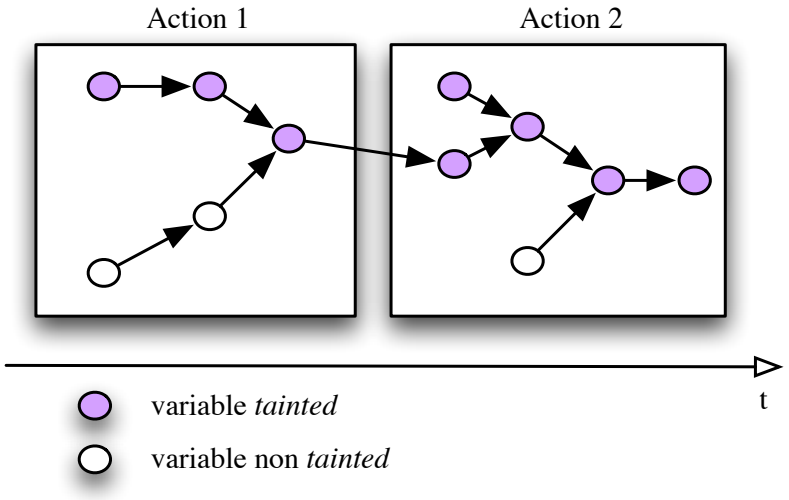


FIGURE 6.3 – Variables causalement dépendantes d'entrées externes

Réduction de l'ensemble des variables Afin de réduire l'ensemble des variables qui doivent être sauvegardées, nous pouvons prendre en considération deux points :
- Tout d'abord, toutes les variables ne sont pas forcément intéressantes du point de vue de la détection d'intrusion : un attaquant a besoin d'interagir avec l'application pour l'attaquer, et il doit donc modifier ses entrées. Les données utilisées par l'application sont altérées par propagation d'erreur. Par conséquent, les variables d'intérêt sont celles qui dépendent des entrées externes de l'application. Cet ensemble de variables peut être défini formellement en utilisant la notion de dépendances causales.
- Même si nous nous restreignons à un ensemble de variables qui dépendent des entrées externes, la quantité de données à conserver en mémoire reste potentiel-

lement importante. Afin de réduire encore cet ensemble, nous allons garder en mémoire l'évolution des variables pendant une fenêtre de temps à déterminer en fonction du domaine d'application (par exemple, dans le cas d'une application web, cela peut correspondre à la fenêtre d'exécution de deux pages dynamiques consécutives exécutées par le même client, comme sur la figure 6.3).

En utilisant la notion de dépendances causales comme définies dans [d'A94], nous définissons qu'une variable v_1 à l'instant t_1 est causalement dépendante d'une variable v_0 à l'instant t_0 (noté $(v_0,\, t_0) \rightarrow (v_1,\, t_1)$) s'il existe un flux d'information de (v_0, t_0) vers (v_1, t_1). L'ensemble des variables qui influencent une variable (v_1, t_1) est appelé le cône de causalité et est noté $cause(v_1,\, t_1) = \{(v_i, t_i) \mid (v_i, t_i) \rightarrow^\star (v_1, t_1)\}$ (où $\rightarrow^\star$ est la fermeture transitive de la relation $\rightarrow$).

Cette définition peut être étendue en utilisant la notion de point d'exécution que nous avons préalablement définie : pour une variable donnée v_0 au point $p_0 = (pc_0, mem_0, t_0)$, avec la variable $v_0 \in V_{pc_0}$, $cause(v_0, p_0) = \{(v, p) \mid (v, p) \rightarrow^* (v_0, p_0)\}$, avec $(v, p) \rightarrow (v_0, p_0) \Leftrightarrow (v, t) \rightarrow (v_0, t_0)$.

Nous définissons alors la notion de variable *tainted* : informellement une variable est *tainted* si elle dépend des entrées externes du programme (par exemple, pour une application web, des entrées utilisateurs, des paramètres de requêtes, ou des données issues d'une base de données). Formellement, cela signifie que le cône de causalité de cette variable à un point p du programme contient une entrée du programme.

On peut alors définir la fonction *tainted* $T_{p,\sigma}$ d'une variable v

$$T_{p,\sigma}(v) = \begin{cases} S_{p,\sigma}(v) \text{ si } v \text{ est } \textit{tainted} \\ \perp \text{ sinon} \end{cases}$$

On obtient ainsi l'ensemble des valeurs qui peuvent être prises par une variable *tainted* pour un compteur ordinal donné :

$$T_{pc}(v) = \bigcup_{\sigma \in \Sigma} [\bigcup_{p \in \sigma | p = (pc, \star, \star)} T_{p,\sigma}(v)].$$

Comme préalablement, on peut définir l'ensemble des variables *tainted* accessibles pour un compteur ordinal donné : nous notons cet ensemble $TV_{pc} = \{v \mid T_{pc}(v) \neq \perp\}$.

Classes d'invariants Une fois l'ensemble des variables déterminé, nous devons générer automatiquement des invariants sur cet ensemble caractérisant le comportement normal de l'application. Comme expliqué précédemment, nous utilisons un logiciel nommé Daikon. Il est nécessaire de lui fournir en entrée les types des invariants que nous recherchons. Nous nous proposons de définir quatre classes d'invariants :

1. une variable est toujours constante à un point d'exécution spécifique ; ce type d'invariant peut être utile pour détecter la violation de l'intégrité de certaines variables, par exemple de valeurs constantes stockées dans la base de données configurant l'application ; ce type d'invariant consiste à trouver que l'ensemble des valeurs de v dans $T_{pc}(v)$, pc étant donné, est un singleton.

2. une variable appartient à un ensemble fini de valeurs à un point d'exécution spécifique ; ce type d'invariant peut être intéressant pour détecter la corruption éventuelle de certaines données fournies par un utilisateur, par exemple un paramètre qui décrit un nombre possible d'options pour l'exécution d'une requête ; ce type d'invariant reflète le fait que toutes les valeurs de v dans $T_{pc}(v)$, pc étant donné, forment un ensemble de cardinal fini.
3. une variable à un point d'exécution est égale à une autre variable à un autre point d'exécution ; ce type d'invariant met en relation des valeurs de variables, ce qui peut permettre de détecter une modification d'une valeur due à une attaque. Par exemple, l'injection SQL décrite précédemment implique la violation d'un invariant de ce type. Ce type d'invariant met en évidence le lien entre les éléments des deux ensembles de valeurs $T_{pc_1}(v_1)$ et $T_{pc_2}(v_2)$ où pc_1 et pc_2 sont deux compteurs ordinaux (pc_1 et pc_2 peuvent être différents ou égaux), et $v_1 \in TV_{pc_1}$ et $v_2 \in TV_{pc_2}$.
4. une relation d'ordre entre deux variables à des points d'exécution différents ($<$ ou $>$) ; de même que pour l'égalité, ce type d'invariant décrit un état normal du programme qui peut être altéré par une attaque suite à une modification de la valeur d'une variable. Ce type d'invariant met en évidence le lien entre les éléments des deux ensembles de valeurs $T_{pc_1}(v_1)$ et $T_{pc_2}(v_2)$ où pc_1 et pc_2 sont deux compteurs ordinaux (pc_1 et pc_2 peuvent être différents ou égaux), et $v_1 \in TV_{pc_1}$ et $v_2 \in TV_{pc_2}$.

En pratique, des invariants très complexes peuvent être calculés, comme par exemple des relations linéaires entre variables. Toutefois, l'expérience nous a montré que les classes d'invariants décrits ci-dessus sont suffisantes pour détecter des attaques traditionnelles connues contre les données. Cela a été vérifié par les évaluations que nous avons menées dans le cadre des applications web. De plus, l'implémentation nous a montré que la recherche de ces quatre classes d'invariants menait à la génération d'un nombre déjà considérable d'invariants.

Il faut en outre noter que les invariants que nous découvrons ne doivent pas seulement être vérifiés par une simple exécution σ, mais par toutes les exécutions de Σ (on rappelle ici que Σ est un ensemble d'exécutions représentatives du comportement normal de l'application considérée). Dans la pratique, les premières exécutions conduisent à la découverte d'un très grand nombre d'invariants. Cet ensemble se réduit ensuite en invalidant des invariants qui ne sont vrais que pour certaines exécutions spécifiques.

6.4 Observation de l'exécution d'un binaire

Afin de générer les traces nécessaires à l'apprentissage inhérent à notre méthode, il nous faut avoir accès aux données internes du programme que l'on observe. Différentes solutions existent qui peuvent être divisées en deux catégories. Une première approche consiste à utiliser les informations de débogage fournies par le système d'exploitation, par exemple grâce à l'appel système `ptrace`. Une deuxième approche consiste à émuler

un processeur contrôlé par le système de détection d'intrusion et à exécuter l'application dans ce contexte. C'est vers cette deuxième approche que nous nous sommes dirigés, en utilisant l'environnement d'instrumentation dynamique de binaire Valgrind [val, NS07].

Valgrind a pour objectif de faciliter l'instrumentation dynamique de binaire. Pour ce faire, il fournit une interface de programmation appropriée et expose au binaire étudié un processeur émulé dans l'espace utilisateur. De manière plus concrète, Valgrind propose l'écriture de *plugins* auxquels il délègue le contrôle de l'application. Ainsi, lorsqu'une application à étudier est exécutée, Valgrind en extrait tout d'abord son bloc de base courant. Il traduit ensuite ce bloc de base en une représentation intermédiaire (IR), constituée d'un ensemble de structures C représentant les différentes instructions du bloc et les différents opérandes de ces instructions. Une fois cette opération achevée, Valgrind transmet le bloc de base ainsi traduit au *plugin*. Le *plugin* peut alors analyser les différentes instructions contenues dans le bloc de base et modifier ce bloc à sa guise en rajoutant ou supprimant des instructions et en y insérant des appels de fonctions C internes à ce *plugin*. Une fois cette opération terminée, le *plugin* retransmet le bloc de base ainsi modifié à Valgrind, qui va le retraduire en code binaire et l'exécuter sur le processeur de la machine hôte. Les instructions ou les appels de fonction insérés par le *plugin* seront alors exécutés dynamiquement. Un système de cache est par ailleurs implémenté pour ne pas avoir à instrumenter plusieurs fois le même bloc de base. L'un des principaux avantages de Valgrind est qu'il tend à rendre ces traitements aussi transparents que possible pour l'application étudiée.

Ainsi, le mécanisme d'observation lié à notre système de détection d'intrusion, a été conçu comme un *plugin* pour Valgrind, et s'appelle *Fatgrind*. Ce prototype est capable dynamiquement de déterminer les données *tainted*, et de collecter leurs valeurs pour obtenir des traces d'exécution utilisables par Daikon.

Génération des traces Le *plugin* Fatgrind que nous avons développé a pour objectif d'extraire dynamiquement les données d'intérêt. Chaque donnée appartenant à cet ensemble est enregistrée afin de générer des traces d'exécution. Pour réaliser cette opération, Fatgrind construit une image de la mémoire du processus exécuté. Cette image de la mémoire conserve une trace des données qui sont *tainted*, et leurs valeurs. Chaque fois qu'un octet est écrit en mémoire, Fatgrind construit le cône de causalité de cet octet (en déterminant quels flux d'information ont été provoqués par l'instruction qui est responsable de l'écriture dans cet octet). Une fois que le cône de causalité a été défini, Fatgrind détermine si l'octet est *tainted* ou non. Dans le cas où il est *tainted*, il est conservé pour une utilisation future. Chaque octet marqué appartenant au cône de causalité est ensuite considéré comme appartenant à l'ensemble des variables d'intérêt et est donc enregistré dans un fichier.

Invariants générés À partir des traces générées, on construit automatiquement les invariants grâce à Daikon. L'approche permet de trouver des invariants pertinents pour détecter des attaques contre des données. Cela a été vérifié sur des attaques réelles. Toutefois, le fait de travailler au niveau binaire ne permet pas d'avoir la connaissance

du type des variables manipulées, de sorte qu'on obtient des invariants de bas niveau qu'il est parfois difficile d'interpréter. C'est pourquoi, dans un deuxième temps, nous avons décidé d'implémenter l'approche sur des programmes interprétés, à savoir des programmes écrits en Ruby.

6.5 Observation de l'exécution d'une application Ruby on Rails

6.5.1 Introduction à Ruby on Rails

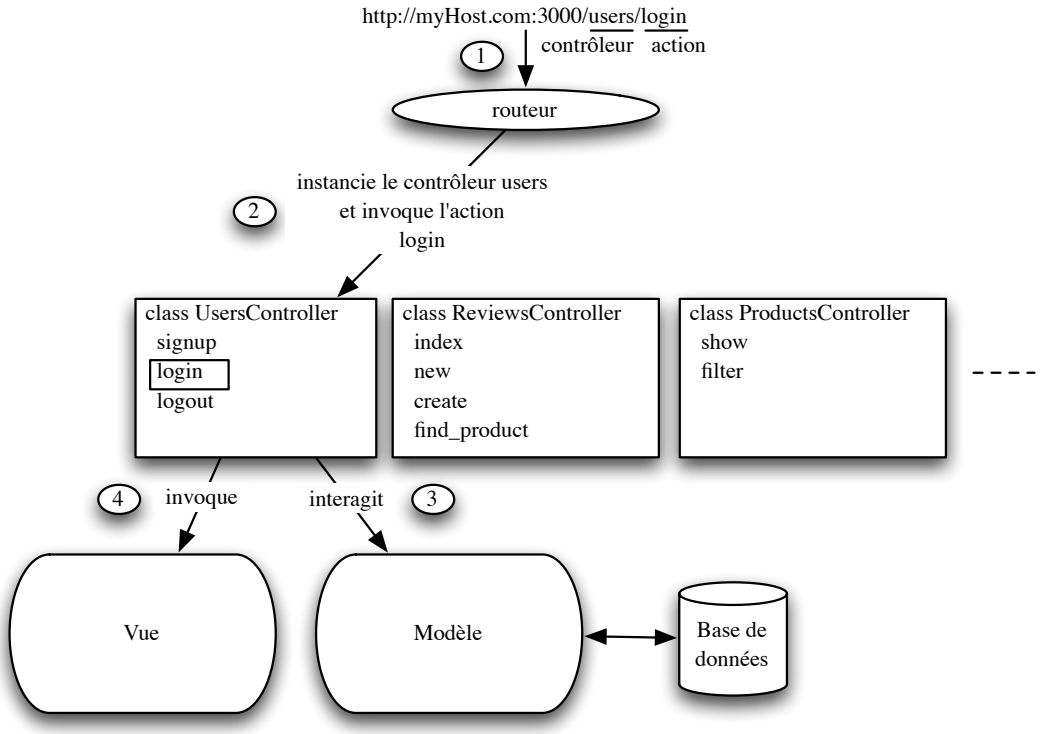


FIGURE 6.4 – Framework Rails dans le contexte de notre application

Ruby on Rails[1] (RoR) est un environment de programmation de type Modèle-Vue-Controleur qui simplifie la réalisation d'applications web. Le choix de RoR comme environnement de prototypage est dû aux capacités d'introspection que le langage Ruby offre au programmeur. En particulier, il est possible de contrôler finement la façon dont le programme se déroule, et d'accéder à l'ensemble de l'état de la mémoire d'un programme n'importe quand pendant son exécution.

1. http ://rubyonrails.org/

Une application RoR typique se compose de plusieurs contrôleurs (comme l'illustre la figure 6.4). Un contrôleur orchestre toutes les actions qui doivent exécutées pour réaliser la fonction qui lui est demandée, en particulier en interagissant avec les vues et les modèles. Une requête HTTP à un serveur web détermine quelle méthode dans quel contrôleur Rails est invoquée. Sur la figure 6.4, la requête indique qu'on invoque la méthode *login* du contrôleur *users*. Toujours sur cette figure, on s'aperçoit que lorsqu'un contrôleur est invoqué, il interagit avec le modèle de données qui définit les informations généralement stockées dans une base de données relationnelle. En utilisant ces données, il génère une vue qui est renvoyée au client, pour être affichée dans le navigateur web. On s'aperçoit que toute requête passe nécessairement par une méthode d'un contrôleur ; nous avons donc choisi d'appliquer notre méthodologie au niveau des contrôleurs.

Ces travaux ont été réalisés dans le cadre d'une application d'e-commerce conçue par Kereval [2] dans le contexte d'une collaboration sur le projet ANR DALI (Design and Assessment of application Level Intrusion detection systems). Cette application nommée *Insecure* est vulnérable aux attaques les plus communément utilisées contre les applications web. En particulier, elle est vulnérable aux attaques suivantes : injections SQL, cross-site scripting, modification de paramètres de requêtes et manipulation de cookies. Cette application a été réalisée sans avoir la connaissance des mécanismes de détection d'intrusion que nous avons mis en place et réciproquement [3], de sorte qu'elle n'est en aucun cas conçue pour mettre en avant la qualité de nos mécanismes de détection.

6.5.2 Phase d'observation

Ruby on Rails permet d'écrire des *plugins* spécifiques, afin d'ajouter des fonctionnalités aux applications. Nous avons utilisé cette capacité pour développer un *plugin* capable d'observer l'état du programme (nommé *trace_vars*) et qui trace et sauvegarde l'état de l'application à chaque instruction exécutée. Ce *plugin* écrit en Ruby est indépendant du code source de l'application. En fait, dans une application RoR, toutes les classes contrôleurs héritent de la classe *ApplicationController*. Ainsi, en modifiant la classe *ApplicationController*, on modifie le comportement de tous les contrôleurs de l'application. Dans la pratique, le *plugin trace_vars* a pour objectif d'interrompre l'exécution à chaque instruction de toute méthode dont la classe hérite de la classe *ApplicationController*, et de sauvegarder l'état de l'application à chaque instruction exécutée par l'interpréteur Ruby. Comme expliqué précédemment, les variables dont la valeur est sauvegardée sont les variables *tainted*, l'interpréteur Ruby étant capable nativement de savoir si une variable dépend d'entrées externes au programme.

2. www.kereval.com
3. Les classes de vulnérabilités sont connues mais pas la manière de les déclencher.

6.5.2.1 Observation de l'état du programme

Afin d'obtenir l'ensemble des variables qui sont disponibles à un point d'exécution donné, nous pouvons utiliser les capacités de méta-programmation de Ruby. En particulier, nous pouvons obtenir toutes les variables d'instances d'un objet et toutes les variables locales d'une méthode en utilisant respectivement les instructions *eval("instance_variables", binding)* et *eval("local_variables", binding)*.

6.5.2.2 Identification des variables *tainted*

Afin de déterminer si une variable est dans l'ensemble des variables qui nous intéressent, nous pouvons construire explicitement son cône de causalité, comme nous l'avons réalisé dans l'observation d'un exécutable binaire. Toutefois, l'interpréteur Ruby offre nativement la notion de variables *tainted*. Pour tout objet Ruby, une méthode *tainted ?* est disponible et permet de savoir si un objet est causalement dépendant d'entrées externes de l'application. Par conséquent, nous utilisons cette fonctionnalité.

6.5.2.3 Nommage des variables dans les traces

Par défaut, l'utilisation de Daikon ne permet pas de générer des invariants sur différentes instances de variables qui évoluent dans le temps. Afin de contourner ce problème, nous introduisons des variables additionnelles, dont le nom est lié au point d'exécution où leur valeur est sauvegardée. Avec cette approche, nous obtenons des règles de nommage des variables de la forme *programCounteur_VariableName*. Le point d'exécution, tel que défini dans ce chapitre comprend le compteur ordinal et un temps logique. Dans l'interpréteur, on peut le modéliser suivant un couple *(ligne, nombreOccurrences)* où *ligne* est la ligne exécutée dans le code source, et *nombreOccurrences* le nombre de fois que cette ligne a été exécutée. Cet artifice, nous permet de relier le point d'exécution du programme à la ligne exécutée dans le code source. Un exemple de trace obtenue est donnée dans la figure 6.5. On voit que ce fichier commence par la déclaration des variables, ainsi que leurs types et leurs attributs. Il se termine par les valeurs observées pour chacune de ces variables à l'exécution. Le programme observé est donné figure 6.2. Dans cette trace, on voit que la variable *params["user"]["login"]* à la première exécution de la ligne 17 vaut la valeur *"evette"*. Par contre, l'interpréteur exécute deux instructions lors de l'exécution de la ligne 19 : un appel de méthode, et une affectation du résultat. La valeur de la variable *session["user"].attributes["login"]* est celle observée durant l'exécution de cette deuxième instruction (d'où le nombre 2 qui suit le numéro de ligne dans le nommage de la variable).

6.5.2.4 Gestion d'une fenêtre temporelle

Afin de générer des invariants qui relient des variables utilisées dans l'exécution de deux requêtes consécutives, nous conservons en mémoire une fenêtre contenant les variables et leurs valeurs rencontrées durant ces deux exécutions. Lorsque nous sauvegardons l'état, nous sauvegardons également l'état de l'exécution précédente. En pra-

```
variable 17_1_@_params["user"]["login"]
  var-kind variable
  dec-type String
  rep-type java.lang.String
  enclosing-var 17_1_@_params["user"]
  comparability 0
variable 19_2_@_session["user"].@attributes["login"]
  var-kind variable
  dec-type String
  rep-type java.lang.String
  enclosing-var 19_2_@_session[user].@attributes
  comparability 0
...
17_1_@_params["user"]["login"]
"evette"
1
19_2_@_session["user"].@attributes["login"]
"evette"
1
```

FIGURE 6.5 – Trace utilisée par Daikon

tique, plusieurs chemins peuvent mener à l'exécution d'une action. Afin de découvrir toutes les relations entre une requête et toutes ses précédentes, il est nécessaire durant la phase d'apprentissage, de parcourir tous ces chemins. Ceci complexifie un peu la phase d'apprentissage, qui toutefois peut être largement automatisée.

6.5.3 Génération d'invariants

Si l'on considère la trace de la figure 6.5, Daikon est capable de générer un des invariants qu'on recherche, à savoir que

$$17\_1\_@\_params[\text{"}user\text{"}][\text{"}login\text{"}] == 19\_2\_@\_session[\text{"}user\text{"}].@attributes[\text{"}login\text{"}]$$

Cet invariant exprime, qu'à la première exécution de la ligne 17, le paramètre *user.login* est égal au résultat de la requête à la base de données stockée dans l'attribut *login* de l'utilisateur en session, lors de la deuxième exécution à la ligne 19. Le fait qu'il s'agit de la deuxième exécution vient du fait qu'en ligne 19, il y a effectivement l'exécution consécutive de deux instructions. La première est la requête à la base de données, et la deuxième l'affection du résultat dans la session.

6.5.4 Vérification des invariants

```
def login
 if request.post?
  if isDefined?('@_params["user"]["login"]') #AutoGenerated
   IDSAsserts.store('17_1_@_params["user"]["login"]',
      @_params["user"]["login"])
  end
  # whole user is stored in the session
  if session[:user] = User.authenticate(
         params[:user][:login],
          params[:user][:password])
  if isDefined?('@_session[:user]["login"]') #AutoGenerated
   IDSAsserts.store('19_2_@_session[:user][:login]',
       @_session[:user][:login])
   assertEqualVar('17_1_@_params["user"]["login"]',
           '19_2_@_session[:user][:login]')
  end
  flash[:notice]  = 'You have been successfully logged in.'
  if session[:user].admin
   redirect_to :controller => '/admin/home',
      :action => 'index'
   else
    redirect_to :controller => '/user/home',
       :action => 'index'
   end
  else
   flash.now[:error] = 'Login failed'
  end
 end
end
```

FIGURE 6.6 – Code Ruby Instrumenté

Une fois que les invariants sont découverts, il ne reste plus qu'à tisser dans le code la vérification de ces invariants sous forme d'assertions exécutables. L'algorithme de tissage est relativement simple, puisqu'il consiste à vérifier l'invariant à la ligne maximum où les variables sont impliquées. Dans l'exemple précédent, il s'agit de la ligne 19. Toutefois, il faut avoir préalablement sauvegardé les valeurs de toutes les variables intervenant dans l'invariant, et ayant été utilisées à des points d'exécution antérieurs. C'est ce qu'on peut remarquer sur la figure 6.6. On sauve ligne 17 la variable *params["user"]["login"]*, et on vérifie après l'authentification que cette sauvegarde est égale à *19_2_@_session[:user][:login]*.

6.5.5 Résultats préliminaires

TABLE 6.1 – Résultats d'instrumentation

Sous-ensemble des fichiers *Insecure*	Nombre initial de lignes	Lignes du code instrumenté	Invariants tissés
application_controller.rb	20	20	0
home_controller.rb	11	537	239
products_controller.rb	26	1503	675
reviews_controller.rb	37	1547	691
users_controller.rb	51	1681	771
...	...	...	...

Les résultats qui sont présentés ici concernent l'application *Insecure*. Cette application est relativement petite (environ un millier de lignes de code Ruby). Sur cette application, on réussit à trouver un nombre très important d'invariants (environ 10000). Ces invariants sont tissés automatiquement dans le code source, ce qui a une incidence non négligeable sur le nombre de lignes de code des différents contrôleurs de l'application (cf. Tableau 6.1). Après quelques mesures rapides, une action d'un contrôleur préalablement à l'instrumentation était exécutée en moyenne en 16.5ms. Après l'instrumentation, le temps moyen d'exécution d'une action est de 125ms. L'impact sur le temps d'exécution est donc non négligeable. Toutefois, d'un point de vue de l'utilisateur, la réponse semble toujours instantanée.

6.6 Évaluation des mécanismes de détection

La phase d'évaluation des mécanismes a été pilotée par le groupe TSF du LAAS-CNRS dans le cadre du projet DALI. Nous ne présenterons ici que deux étapes de cette évaluation : la première étape consiste en l'évaluation du comportement de l'IDS en présence d'un trafic sain. Ces résultats sont donnés dans le tableau 6.2. La deuxième étape consiste en l'évaluation de l'IDS en présence d'attaques. Elle a consisté à trouver automatiquement des points d'injection pour des injections SQL, et à former automatiquement des requêtes mal formées qui pourraient exploiter une vulnérabilité de type SQL injection. Le lecteur est invité à consulter [DAA+11] pour plus de détails sur la manière utilisée pour découvrir automatiquement les vulnérabilités.

Nombre de requêtes	Nombre d'alarmes soulevées	Nombre d'invariants violés
1623	20	4

TABLE 6.2 – Nombre d'invariants violés en trafic sain

Nombre de requêtes générées	Nombre d'attaques réussies	Nombre d'alarmes émises	faux positifs	faux négatifs
13320	11	11	0	0

TABLE 6.3 – Situation d'attaques

On s'aperçoit qu'en présence de trafic sain, pour 1623 requêtes, 20 alarmes ont été levées, correspondant à 4 invariants incorrects. Le fait que des invariants incorrects existent est dû au fait que l'apprentissage est incomplet, et que des invariants vraisemblables mais en réalité faux n'ont pas été invalidés lors de la phase d'apprentissage. Toutefois, ces invariants, une fois identifiés, peuvent être facilement invalidés, en complétant l'apprentissage qui peut être mené de manière incrémentale.

En présence d'attaques (cf. Tableau 6.3), 13320 requêtes mal formées ont été envoyées à l'application, ce qui a généré 11 alertes. Après vérification, ces 11 alertes correspondent aux 11 seules requêtes qui ont réussi à exploiter une vulnérabilité. Dans cette phase, il n'y a donc eu ni faux positif, ni faux négatif.

6.7 Travaux connexes

Ces travaux se fondent sur un outil particulier : Daikon. Il a déjà largement été utilisé dans de nombreux projets, mais nous noterons particulièrement son utilisation en détection d'intrusion dans le cadre d'un outil nommé *Swaddler* [CBFV07]. Comme dans notre approche, Daikon est utilisé pour calculer un comportement normal invariant d'une application web, cette fois écrite en PHP. Toutefois, l'utilisation qui est faite de Daikon est très classique, et les invariants recherchés portent sur les entrées ou sorties de fonctions, contrairement à notre approche où nous atteignons un niveau de granularité beaucoup plus fin, au niveau des instructions. Notre approche permet de découvrir beaucoup plus d'invariants, et donc potentiellement de détecter plus d'attaques. Dans [FCKV10], les mêmes auteurs utilisent une approche similaire pour des applications écrites en Java, afin cette fois-ci de découvrir des vulnérabilités dans les applications web fondées sur les JSP. Ce qui est intéressant dans ces nouveaux travaux est la capacité d'invalider des invariants potentiellement faux (à cause d'une phase d'apprentissage incomplète) en analysant statiquement le programme source.

6.8 Conclusion

Les résultats obtenus dans le cadre de ces études sont prometteurs. Non seulement les implémentations proposées montrent la faisabilité de l'approche, mais de plus, les résultats en terme de détection sont encourageants. De surcroît, les deux approches statique (cf. chapitre 5) et dynamique (cf. chapitre 6) permettent de générer des in-

variants qui sont tout à fait complémentaires. Les outils à notre disposition n'ont pas permis d'appliquer les deux approches à un même langage, ce qui est dommage, car cela aurait permis de mesurer plus aisément leur complémentarité.

Le principal inconvénient de l'approche dynamique réside dans la difficulté de mettre en place la phase d'apprentissage. Il faut en effet être capable de couvrir l'ensemble des comportements normaux de l'application, ce qui risque d'être un objectif difficilement atteignable dans le cas d'une application de grande taille. Toutefois, il faut relativiser cet inconvénient, car il est possible, comme dans le domaine du test, de générer automatiquement des tests permettant de couvrir ces comportements de façon relativement exhaustive.

Chapitre 7

Conclusion Générale

Dans ce mémoire, nous avons présenté quatre contributions scientifiques qui ont un point commun : l'utilisation de techniques de détection d'erreur appliquées à la détection d'intrusion. Les deux premières contributions reposent sur l'utilisation de la diversification fonctionnelle, et proposent de détecter des erreurs soit sur les entrées-sorties réseau de services (le système d'intrusion s'apparente donc à un système de détection au niveau réseau ou NIDS), soit sur les entrées/sorties au niveau des systèmes d'exploitation (le système d'intrusion se positionne donc au niveau de chaque nœud, et on a donc affaire à un Host-based IDS, ou HIDS). Les deux dernières contributions reposent sur la technique des contrôles de vraisemblance très utilisée en sûreté de fonctionnement pour détecter des erreurs dans une application. Ces deux travaux reposent sur la génération automatique de ces contrôles, via la découverte d'invariants au sein du logiciel, que ce soit de manière statique ou de manière dynamique. Ces deux approches, appliquées au sein du logiciel, forment un complément idéal aux NIDS et HIDS présentés dans les deux premières contributions.

L'ensemble de ces contributions s'intéresse à un service centralisé, et cherche à détecter des intrusions contre ce service. Une piste de travail future est d'adapter ces travaux au cas d'applications distribuées. Bien que quelques travaux se soient intéressés à ces types d'applications, il nous semble que les propriétés sur lesquelles ils reposent sont bien trop fortes (présence d'horloges globales par exemple). Il nous apparait que la notion d'invariant définie dans l'approche centralisée peut être étendue dans le cas d'applications distribuées, et peut donc faire l'objet de nouveaux travaux à venir. Dans le cadre de la toute nouvelle équipe INRIA CIDRE, ce type de travaux a été défini comme un axe de recherche de l'équipe.

Bibliographie

[AAC+03] A. Adelsbach, D. Alessandri, C. Cachin, S. Creese, Y. Deswarte, K. Kursawe, J.-C. Laprie, D. Powell, B. Randell, J. Riodan, P. Ryan, R. J. Stroud, P. Verissimo, M. Waidner, and A. Wespi. Conceptual model and architecture of MAFTIA. MAFTIA deliverable d21, LAAS-CNRS and University of Newcastle upon Tyne, January 2003.

[ABEL03] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. A theory of secure control-flow. In *Proceedings of the International Conference on Formal Engineering Methods (ICFEM'2003)*, 2003.

[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*, pages 340–353, New York, NY, USA, 2005. ACM Press.

[AC77] Algirdas Avizienis and L. Chen. On the implementation of N-version programming for software fault tolerance during execution. In *Proceedings of the IEEE International Computer Software and Applications Conference (COMPSAC 77)*, pages 149–155, Chicago, IL, November 1977.

[ACR+08] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with wit. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 263–277, 2008.

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1) :11–33, 2004.

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

[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.

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

[CBFV07] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna. Swaddler : An Approach for the Anomaly-based Detection of State Violations in Web

Applications. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 63–86, Gold Coast, Australia, September 2007.

[CC77] 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.

[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*, volume 7, page 11, 2006.

[CEA] CEA. Frama-c, framework for modular analysis of c.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1978.

[CS03] Pierre-Antoine Champin and Christine Solnon. Measuring the similarity of labeled graphs. In *Proceedings of the 5th International Conference on Case-Based Reasoning (ICCBR 2003)*, pages 80–95, Trondheim, Norway, June 2003.

[CXS+05] S. Chen, J. Xu, E. Sezer, P. Gauriar, and R. Iyer. Non-control-data attacks are realistic threats. In *Usenix Security Symposium*, pages 177–192, 2005.

[d'A94] Bruno d'Ausbourg. Implementing secure dependencies over a network by designing a distributed security subsystem. In *Proceedings of the Third European Symposium on Research in Computer Security (ESORICS'94)*, pages 247–266, 1994.

[DAA+11] A. Dessiatnikoff, R. Akrout, E. Alata, M. Kaaniche, and V. Nicomette. A clustering approach for web vulnerabilities detection. In *Proceedings of Pacific Rim International Symposium on Dependable Computing*, pages 194–203, Los Alamitos, CA, USA, 2011.

[DDW99] Hervé Debar, Marc Dacier, and Andreas Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 1999.

[DMTT11] Jonathan-Christofer Demay, Frederic Majorczyk, Eric Totel, and Frederic Tronel. Detecting illegal system calls using a data-oriented detection model. In *Proceedings of the 26th IFIP TC-11 International Information Security Conference (IFIP SEC2011)*, June 2011.

[EPG+07] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69 :35–45, 2007.

[FCKV10] Viktoria Felmetsger, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Toward automated detection of logic vulnerabilities in web

applications. In *19th USENIX Security Symposium*, Washington, DC, August 2010.

[Gra89] P. Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 30 :165–190, 1989.

[Gra91] P. Granger. Static analysis of linear congruence equalities among variables of a program. In *TAPSOFT'91*, pages 169–192, 1991.

[GRRV03] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, and M. 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.

[GRS04a] 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*, pages 318–329, 2004.

[GRS04b] Debin Gao, Michael K. Reiter, and Dawn Song. On gray-box program tracking for anomaly detection. *USENIX Security Symposium*, 2004.

[GRS05] Debin Gao, Michael K. Reiter, and Dawn Song. Behavioral distance for intrusion detection. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID 2005)*, pages 63–81, Seattle, WA, September 2005.

[GRS08] Debin Gao, Michael K. Reiter, and Dawn Song. Beyond output voting : Detecting compromised replicas using hmm-based behavioral distance. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, July 2008.

[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.

[HTMM09] Guillaume Hiet, Valerie Viet Triem Tong, Ludovic Me, and Benjamin Morin. Policy-based intrusion detection in web applications by monitoring java information flows. *Int. J. Inf. Comput. Secur.*, 3(3/4) :265–279, 2009.

[JRC+02] James E. Just, James C. Reynolds, Larry A. Clough, Melissa Danforth, Karl N. Levitt, Ryan Maglich, and Jeff Rowe. Learning unknown attacks - a start. 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 158–176, Zurich, Switzerland, October 2002. Springer.

[Kar76] M. Karr. Affine relationships among variables of a program. In *Acta Informatica*, pages 133–151, 1976.

[KBA02] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the Usenix Security Symposium*, pages 191–206, 2002.

[KFL94] Calvin Ko, George Fink, and Karl N. Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of*

the 10th Annual Computer Security Applications Conference (ACSAC'94), pages 134–144, Orlando, FL, December 1994.

[KKP+81] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and its use in optimization. In *Proceedings of the Eighth ACM Symposium on Principles of Programming Languages*, pages 207–218, 1981.

[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.

[LABK90] Jean-Claude Laprie, Jean Arlat, Christian Béounes, and Karama Kanoun. Definition and analysis of hardware-and-software fault-tolerant architectures. *IEEE Computer*, 23(7) :39–51, July 1990.

[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.

[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.

[NS07] Nicolas Nethercote and Julian Seward. Valgrind : A framework for heavy-weight dynamic binary instrumentation. In *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, 2007.

[Ran75] Brian Randell. System structure for software fault tolerance. In *Proceedings of the International Conference on Reliable software*, pages 437–449, Los Angeles, CA, April 1975.

[SND08] Ayda Saidane, Vincent Nicomette, and Yves Deswarte. The design of a generic intrusion-tolerant architecture for web servers. *IEEE Transactions on Dependable and Secure Computing*, 5(2), April-June 2008.

[TBK+03] Chin-Yang Tseng, Poornima Balasubramanyam, Calvin Ko, Rattapon Limprasittiporn, Jeff Rowe, and Karl N. Levitt. A specification-based intrusion detection system for AODV. In *2003 ACM Workshop on security of Ad Hoc and Sensor Networks (SASN '03)*, pages 125–134, Fairfax, VI, October 2003.

[TMM05] Eric Totel, Frédéric Majorczyk, and Ludovic Mé. COTS diversity based intrusion detection and application to web servers. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID 2005)*, pages 43–62, Seattle, WA, september 2005.

[TSB+05] Chinyang Henry Tseng, Tao Song, Poornima Balasubramanyam, Calvin Ko, and Karl Levitt. A specification-based intrusion detection model for

olsr. In *Proceedings of 8th International Symposium on Recent Advances in Intrusion Detection (RAID '2005)*, september 2005.

[VA06] Ramtilak Vemu and Jacob A. Abraham. Ceda : Control-flow error detection through assertions. In *Proceedings of the 12th IEEE International On-Line Testing Symposium (IOLTS'06)*, 2006.

[val] Valgrind. www.valgrind.org.

[VRKK03] Giovanni Vigna, William Robertson, Vishal Kher, and Richard A. Kemmerer. A stateful intrusion detection system for world-wide web servers. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC 2003)*, pages 34–43, Las Vegas, NV, December 2003.

[Wei82] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4) :352–357, 1982.

[ZMB03] 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.