N° ORDRE 3827
de la thèse

Thèse

présentée

DEVANT L'UNIVERSITÉ DE RENNES 1

pour obtenir le grade de :

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : INFORMATIQUE

PAR

Frédéric Majorczyk

Équipe d'accueil (du doctorant)	**SSIR** (Sécurité des systèmes d'informations et réseaux) Supélec avenue de la Boulaie F-35576 Cesson-Sévigné
École Doctorale	**Matisse**
Composante universitaire (du directeur de thèse) UFR	**SUPÉLEC**

"Détection d'intrusions comportementale par diversification de COTS : application au cas des serveurs web"

Soutenue le 3 décembre 2008 devant la commission d'examen

Composition du jury :

- Monsieur César Viho ..... Président du Jury
- Monsieur Salem Benferhat ..... Rapporteur
- Monsieur Yves Deswarte ..... Rapporteur
- Monsieur Éric Totel ..... Co-directeur de thèse
- Monsieur Ludovic Mé ..... Directeur de thèse

« *I need a thesis bailout.* »
The unnamed phd student, phdcomics.

Remerciements

Je remercie tout d'abord Salem Benferhat et Yves Deswarte d'avoir bien voulu accepter la charge de rapporteur et d'avoir émis des commentaires intéressants qui ont permis d'améliorer ce mémoire. Je remercie également César Viho d'avoir accepté la charge de présider le jury lors de ma soutenance.

Je remercie évidemment Éric Totel et Ludovic Mé pour m'avoir encadré et supporté (dans tous les sens du terme) pendant ces parfois trop longues années.

Je remercie les enseignants et le personnel de Supélec pour m'avoir offert et offrir un cadre de travail agréable à tous les thésards. Un grand merci à Valérie pour son aide précieuse (et son papyrus) lors de ma dernière année de thèse !

Un grand merci à tous les thésards/post-docs/stagiaires cotoyés pendant ces 4 ans à Supélec et ailleurs ; en vrac et en espérant n'oublier personne : Whylmiark, Fanch, Neerd, Nico, Nico (l'autre) et Nico (le troisième), Mihai, Marius, Yasser, Tosh, Jonathan, Olivier (le coco), Mehdi, Dali, Domtom, Julien, Loïg, Asmaa, Ayda, aux footeux de la fac et de l'enstb et je suis sûr d'oublier des gens... (pardon aux familles...).

Un grand merci aux étudiants et maintenant ex-étudiants cotoyés tous les jours (ou presque) à la résidence (sans vous, les soirées auraient été bien différentes ! Merci ! et merci à ceux qui m'ont poussé à finir !). Pareil en vrac et pardon à ceux que j'oublie : Colmi, Guiniol, Coyote, Crako (et oui, t'es passé à la rez), Scelmorp, JP3, Popol, Rustik, Oz, Strahd, Dex, Popo, Sermac, Chouf, Gege24, Joelf, Seb, Dromi, Isa, Marco, Aurélien, Gagou, Marie, HC, David, Phyce, Guitou, Legolas, Yeti, SK, Carl, Viking, micro-Stef, Bertrand, Maxime (du rezo) et Maxime (ou Mathieu), Flo, Léo, Christophe, Alexis, Thomas, Maël, Gaétan et les autres !

Un grand merci à tous les amis rencontrés au hasard de mes pérégrinations : Silozius (merci d'avoir relancé la rédaction !) et Geneviève, Buju, Martin, Thomas, Mathieu, Lénaïc, Een, Power, Akbar, Stilgar, Jérôme, Étienne, Dahab, Elena. Un grand merci aux amis de longue date et à mes frères : Alex et David, Julien, Matus, Tho, Pierre, Edith, Jérôme, JA, Philippe.

Merci au bananier du niveau 5 !

Merci à Primordial et à Opeth (et au heavy metal en général \m/) !

Merci à mes parents !

Table des matières

Table des figures

Liste des tableaux

Introduction

« *Universe, n. : The problem* »

L'informatique et en particulier l'Internet jouent un rôle grandissant dans notre société. Un grand nombre d'applications critiques d'un point de vue de leur sécurité sont déployées dans divers domaines comme le domaine militaire, la santé, le commerce électronique, etc. La sécurité des systèmes informatiques devient alors une problématique essentielle tant pour les individus que pour les entreprises ou les états.

Pour chaque système informatique, *une politique de sécurité* doit donc être définie pour garantir les propriétés de sécurité qui doivent être rendues par ce dernier. Cette politique s'exprime par des règles fixant trois objectifs distincts :

- **la confidentialité** c'est-à-dire la non-occurrence de divulgations non-autorisées de l'information ;
- **l'intégrité** c'est-à-dire la non-occurrence d'altérations inappropriées de l'information ;
- **la disponibilité** c'est-à-dire le fait d'être prêt à l'utilisation.

Dans cette thèse, nous entendons par intrusion, une violation d'un de ces trois objectifs. Plusieurs approches ont été définies pour s'assurer que la politique de sécurité définie pour un système informatique est bien respectée. Elle peut en effet être contournée par un utilisateur malveillant ou plus simplement une faute de conception peut être à l'origine d'une violation de celle-ci.

La première approche s'appuie sur des mécanismes préventifs : il s'agit alors de mettre en place des dispositifs capables d'empêcher toute action qui entraînerait une violation de la politique de sécurité. Cependant l'expérience montre qu'il est impossible de construire un système entièrement sûr pour des raisons techniques (granularité du contrôle d'accès par exemple) ou pratiques (la mise en place d'un système préventif occasionnerait une gêne trop importante pour les utilisateurs par exemple). Il est en outre très difficile de développer un logiciel complexe exempt de fautes de conception dont certaines peuvent être exploitées afin de produire un non-respect de la politique de sécurité.

Acceptant ce fait, une deuxième approche pour traiter les intrusions consiste à détecter les violations de la politique de sécurité et à les signaler aux administrateurs pour qu'ils puissent prendre les mesures nécessaires pour remédier aux problèmes éventuels qu'ont pu générer ces violations. La détection d'intrusions est fondée sur l'analyse à la volée ou en différé de ce qui se passe sur le système.

Une troisième approche, la tolérance aux intrusions, vise à ce que le service reste assuré et que la politique de sécurité du système global reste inviolée même en présence d'intrusions dans certains composants du système. Des intrusions peuvent affecter certains composants du système mais les propriétés de confidentialité, d'intégrité et de disponibilité du système global doivent être vérifiées.

Les travaux que nous présentons dans cette thèse s'intègrent de manière essentielle dans le domaine de la détection d'intrusions et permettent de plus une certaine tolérance aux intrusions.

En détection d'intrusions, deux types d'approche sont utilisées principalement : l'approche par signature (*misuse detection*) et l'approche comportementale (*anomaly detection*).

L'approche par signature consiste à définir des scénarios d'attaque et rechercher des traces de ces scénarios, par exemple dans les fichiers d'audit du système. Cette approche pose certains problèmes notamment celui de la détection de nouvelles attaques (c'est-à-dire des attaques dont la signature n'est pas encore dans la base et qui nécessitent la mise à jour de la base de signatures et ce de manière extrêmement fréquente) ou celui de la détection d'attaques inconnues.

Le principe de base de l'approche comportementale est de construire un modèle de référence du comportement de l'entité surveillée (utilisateur, machine, service, application) auquel on peut comparer le comportement observé. Si ce dernier est trop éloigné de la référence, une alerte est émise pour signaler l'anomalie. Les techniques classiques fondées sur l'approche comportementale proposent un modèle de référence construit de manière explicite. Cependant il n'est pas simple de définir ce qui est représentatif du comportement à modéliser et les systèmes de détection d'intrusions (IDS pour *Intrusion Detection Systems*) fondés sur cette méthode génèrent un grand nombre de fausses alertes.

Face à ces difficultés, il peut être intéressant de rechercher une approche où le modèle de référence est implicite. L'approche que nous proposons s'inspire d'une méthode classique en sûreté de fonctionnement : la diversification fonctionnelle et plus spécifiquement la programmation N-versions. La diversification fonctionnelle est une technique de tolérance aux fautes de conception qui s'appuie sur la comparaison, par un décideur, des résultats fournis par des mises en œuvre dissemblables (ou variantes) d'une même fonction. Dans le cas de la programmation N-versions, les variantes sont appelées des versions et le décideur effectue un vote sur les résultats de toutes les versions. Les versions ont la même spécification mais sont de conception différente (équipes de dévelop-

peurs différentes, utilisation de plates-formes différentes, etc.). Ces différences dans la conception assurent avec une probabilité assez élevée que les fautes présentes dans les variantes seront indépendantes c'est-à-dire qu'elles n'auront pas de causes communes. Cette technique a cependant un coût élevé puisqu'il est nécessaire de développer chaque version indépendamment. Elle a ainsi été essentiellement utilisée dans des systèmes critiques du point de vue de la sécurité-innocuité tels que l'aéronautique ou le ferroviaire. Il est possible d'utiliser des COTS (*Components-Off-The-Shelf*) en lieu et place des versions spécifiquement développées. Nous appelons cette technique la diversification de COTS. Ce choix est plus économique mais pose certaines difficultés qu'il faut prendre en compte et sur lesquels nous reviendrons.

La diversification de COTS peut être appliquée au domaine de la sécurité pour assurer des propriétés de détection d'intrusions et de tolérance aux intrusions. C'est l'approche que nous avons poursuivie lors de cette thèse. Plusieurs autres projets et travaux de recherche sont également fondés sur cette idée, soit dans une optique de tolérance aux intrusions, soit dans une optique de détection d'intrusions. Ces projets et travaux peuvent être classés, suivant les éléments de comparaison choisis, en deux types d'approches : les approches de type boîte noire et les approches de type boîte grise. Les approches de type boîte noire comparent le comportement des COTS au niveau de sorties standardisées. Les approches de type boîte grise comparent le comportement interne des COTS.

Les projets de type boîte noire se sont essentiellement consacrés à la tolérance aux intrusions de leur architecture. Ces projets, à l'exception du projet BASE [CRL03], utilisent d'autres IDS pour assurer la détection et la détection d'intrusions ne repose que très peu sur la diversification de COTS. Très peu d'évaluations ont été faites sur les taux de couverture. De la même manière, très peu d'évaluations ont été réalisées sur les faux positifs générés par l'architecture que ce soit de manière théorique ou pratique, excepté dans le cadre du projet BASE, où les auteurs proposent un modèle abstrait permettant de masquer les différences de spécification. Nous proposons dans cette thèse une première méthode de détection d'intrusions qui suit une approche de type boîte noire. Nous avons cherché à évaluer l'efficacité de l'approche par diversification en termes de faux positifs notamment.

Les projets suivant une approche de type boîte grise peuvent être classés en deux catégories : les projets utilisant des techniques de diversification automatique et les travaux de Gao proposant l'utilisation de diversité naturelle. Les projets utilisant des techniques de diversification automatique ont un taux de couverture limité à certaines classes d'attaques. L'utilisation de diversité naturelle laisse espérer un taux de couverture de détection plus important. La seconde méthode de détection que nous proposons dans cette thèse se rapproche des travaux de Gao [Gao07]. Cependant, contrairement à ces travaux, notre approche ne repose pas sur l'apprentissage des correspondances entre les séquences d'appels système mais sur une abstraction sous la forme de graphes de

flux d'informations. Cette abstraction permet d'apporter plus de sémantique à la détection et peut fournir une aide à un premier diagnostic, facilitant ainsi le travail de l'administrateur de sécurité.

Dans cette thèse, nos contributions sont donc les suivantes :
- La prise en compte dans notre modèle général de détection d'intrusions des spécificités liées à l'utilisation de COTS en lieu et place de versions spécifiquement développées et la proposition de deux solutions pour parer aux problèmes induits par ces spécificités.
- Une étude théorique de la diversification de COTS pour la détection et la tolérance aux intrusions.
- Deux méthodes de détection d'intrusions fondées sur une architecture de type N-versions : l'une suivant une approche de type boîte noire et l'autre suivant une approche de type boîte grise. Notre méthode de type boîte grise peut aider l'administrateur de sécurité à effectuer un premier diagnostic des alertes.
- Deux IDS pour serveurs web fondés chacun sur une des ces méthodes et l'évaluation pratique de la pertinence et de la fiabilité de ces deux IDS.

Le mémoire est organisé de la manière suivante.

Le chapitre 1 décrit les grands domaines qui soutiennent cette thèse et les travaux similaires aux nôtres. Il est divisé en trois parties. La première partie présente les deux grandes approches de détection d'intrusions et également la détection d'intrusions web qui sera le domaine d'application de nos propositions. La deuxième partie décrit les grands principes de la sûreté de fonctionnement qui sont la base théorique de notre approche. La troisième partie présente les travaux de recherche similaires aux nôtres : les approches de type boîte noire et de type boîte grise.

Le chapitre 2 présente le cœur de notre travail. Nous présentons en détail l'architecture de détection d'intrusions que nous avons utilisée et les propriétés qui découlent de l'utilisation de COTS et d'un modèle de fautes intentionnelles où l'attaquant a la possibilité de contrôler l'erreur que son intrusion produit. Nous décrivons les deux méthodes de détection d'intrusions que nous avons proposées : l'approche de type boîte noire est très proche du modèle général de détection alors que l'approche de type boîte grise s'en distingue bien plus. Cette méthode repose sur un calcul de similarité entre des graphes de flux d'informations. Ces graphes ont une sémantique très forte et les différences entre les graphes des différents serveurs peuvent servir d'aide à un premier diagnostic des alertes.

Le chapitre 3 détaille l'application aux serveurs web et donne des indications sur les implémentations de nos deux IDS pour serveurs web. Nous présentons d'abord rapidement le protocole HTTP qui est l'un des protocoles qui soutiennent l'Internet. Nous présentons la manière dont nous avons appliqué les concepts présentés au chapitre 2 dans le cadre des serveurs web pour l'approche de type boîte noire et pour l'approche de type boîte grise tout en essayant de

présenter les limitations techniques auxquelles nous avons été confrontés qui peuvent diminuer le taux de couverture de l'approche et/ou augmenter le taux de faux positifs.

Le chapitre 4 présente les résultats des tests que nous avons menés sur les deux IDS développés pour les serveurs web et présentés dans le chapitre 3. Nous présentons d'abord dans ce chapitre l'environnement de test et les jeux de données que nous avons utilisés : attaques et trafic « normal ». Nous présentons ensuite les résultats de détection obtenus pour nos deux IDS en évaluant leur fiabilité et leur pertinence. Nous détaillons par la suite deux exemples d'aide au diagnostic que peut apporter notre approche de type boîte grise. Finalement, nous évaluons les performances des nos IDS.

Enfin, notre conclusion constitue un récapitulatif général du travail accompli : méthodes de détection proposées, leur application dans le cadre des serveurs web et les résultats en terme de détection des deux IDS développés. Nous terminons par quelques perspectives pour les suites à donner à ces travaux.

Chapitre 1

État de l'art

« De toutes les misères humaines voici la pire de toutes :
de tant savoir de choses et de si peu pouvoir les maîtriser. »
Hérodote

Ce chapitre présente les travaux antérieurs liés aux domaines de cette thèse que sont la détection d'intrusions et la sûreté de fonctionnement.

Nous allons d'abord présenter les deux grandes approches en détection d'intrusions : l'approche par signature et l'approche comportementale. Les prototypes implémentant les méthodes de détection d'intrusions proposées dans cette thèse ayant été développés pour des serveurs web, nous détaillons ensuite des méthodes de détection d'intrusions et IDS spécifiques aux intrusions web.

La deuxième section de ce chapitre présente, d'abord, certaines notions fondamentales en sûreté de fonctionnement que sont les notions de faute, erreur et défaillance et le concept de tolérance aux fautes. Dans la suite de cette section, nous présentons, de manière générale, un moyen d'assurer la sûreté de fonctionnement d'un système : la diversification fonctionnelle, et ensuite, plus précisément, une technique spécifique de diversification fonctionnelle : la programmation N-versions. Ce chapitre se termine par une analyse des concepts du domaine de la sécurité dans le domaine de la sûreté de fonctionnement.

La troisième section de ce chapitre présente les principes de la détection d'intrusions et de la tolérance aux intrusions par diversification fonctionnelle. Nous présentons ensuite les divers travaux fondés sur ces principes et similaires en cela aux nôtres tout en positionnant les deux méthodes proposées dans cette thèse par rapport à ces travaux.

1.1 Détection d'intrusions

La détection d'intrusions a été introduite en 1980 par J.P. Anderson qui a été le premier à montrer l'importance de l'audit de sécurité [And80] dans le but de détecter les éventuelles violations de la politique de sécurité d'un système. Anderson définit une intrusion comme une violation de la politique de sécurité

du système, c'est-à-dire une violation d'une des propriétés de confidentialité, d'intégrité ou de disponibilité du système. Nous différencions les notions d'attaques et d'intrusions : une intrusion est une violation de la politique de sécurité alors qu'une attaque est une tentative (effective ou non) de violer la politique de sécurité du système.

Le but de la détection d'intrusions est de signaler les intrusions ou les attaques, suivant la conception de l'IDS, à l'administrateur de sécurité pour que celui-ci puisse prendre les mesures de réaction adéquates, comme remettre le système dans un état sûr. La recherche en détection d'intrusions s'oriente également vers la réponse automatique aux intrusions [Tho07] : en plus de prévenir l'administrateur de sécurité, le système de détection d'intrusions (on parle alors généralement d'IPS, *Intrusion Prevention System* ou système de prévention d'intrusion) prend des mesures pour bloquer l'intrusion : couper la connexion TCP et changer les règles du pare-feu pour bloquer l'adresse IP de l'attaquant pour un IPS réseau par exemple.

Les méthodes de détection d'intrusions reposent sur l'observation d'un certain nombre d'événements et sur l'analyse de ceux-ci.

Il s'agit premièrement de collecter les informations que l'on souhaite analyser. Ces informations proviennent des fichiers journaux du système, d'applications spécifiques (tels que des serveurs web, serveurs ftp, serveurs de courriers électroniques, etc...) ou de sondes mises en place par les outils de détection d'intrusions tels des « sniffers » réseau, des modules spécifiques à certaines applications ou au système d'exploitation. On distingue généralement deux grands types de sources de données et l'on peut classer les IDS suivant leurs sources : les flux réseau (*network-based IDS ou NIDS*) et les données système ou applicatives (*host-based IDS ou HIDS*). Suivant les sources de données utilisées, il est possible de détecter certaines intrusions ou non : une attaque par déni de service par *syn flooding* [Edd07] est plus facilement détectable au niveau réseau qu'au niveau applicatif. Certains IDS utilisent des données provenant des deux types de sources pour augmenter leur capacité de détection.

Les données récoltées doivent ensuite être analysées pour y rechercher des traces d'intrusions. Cette analyse peut se faire de plusieurs manières : après les faits, de manière quasi temps-réel ou bien en temps réel. Deux approches principales de détection d'intrusions que nous décrirons par la suite ont été proposées : l'approche par signature (*misuse detection*) et l'approche comportementale (*anomaly detection*). Ces approches présentent différents avantages et inconvénients ; il a été proposé relativement rapidement dans la recherche en détection d'intrusions de combiner ces deux approches pour augmenter leur taux de détection [Lun88, HK88].

L'évaluation et la comparaison des systèmes de détection d'intrusions est un problème en soi de par la diversité des sources de données possibles et la représentativité des données utilisées lors des tests notamment. Une vue d'ensemble des problèmes et des différents projets de plate-formes de tests peut être trouvée dans [MHL+03]. Les systèmes de détection d'intrusions sont évalués

traditionnellement suivant deux critères :

- **La fiabilité** de l'IDS : toute intrusion doit effectivement donner lieu à une alerte. Une intrusion non signalée constitue une défaillance de l'IDS, appelée faux négatif. La fiabilité d'un système de détection d'intrusions est liée à son taux de faux négatifs (c'est-à-dire le pourcentage d'intrusions non-détectées), qui doit être le plus bas possible.
- **La pertinence** des alertes : toute alerte doit correspondre à une intrusion effective. Toute « fausse alerte » (appelée également faux positif) diminue la pertinence de l'IDS. Un bon IDS doit présenter un nombre de faux positifs aussi bas que possible.

Il ne suffit pas de détecter correctement les intrusions ; il faut surtout éviter de lever trop de fausses alertes. Si la pertinence des alertes est trop faible, par exemple si plus de 50% des alertes sont des faux positifs (50% étant un chiffre prudent), l'administrateur de sécurité risque de considérer toutes les alertes comme des faux positifs [Axe00a], de ne pas les analyser et de ne pas prendre des mesures de protection au cas où l'alerte ne serait pas un faux positif. Dans ce cas, le système de détection d'intrusions devient inutile.

Dans la suite, nous allons décrire les deux principales approches de détection d'intrusions puis les IDS spécifiques aux serveurs web

1.1.1 Détection d'intrusions par signature et détection d'intrusions comportementale

Les systèmes de détection d'intrusions peuvent être classés suivant leur approche d'analyse des données. Deux grandes approches ont été proposées dans la littérature : la détection d'intrusions par signature et la détection d'intrusions comportementale. Ces deux approches s'opposent dans leur principe de détection : l'approche par signature se fonde sur la recherche de traces d'attaques ou d'intrusions alors que l'approche comportementale recherche les déviations du comportement de l'entité observée par rapport à un modèle du comportement normal de cette entité. Bien que la première approche proposée par Anderson en 1980 [And80] soit de type comportementale, nous allons d'abord présenter les approches par signature qui ont les faveurs des industriels de la sécurité mais présentent des inconvénients, inhérents à l'approche, difficilement contournables.

1.1.1.1 Détection d'intrusions par signature

Principe Les systèmes de détection d'intrusions par signature fondent leur détection sur la reconnaissance, dans le flux d'évènements générés par une ou plusieurs sondes, de signatures d'attaques qui sont contenues dans une base de signatures. Une signature est un motif, dans le flux d'événements, de scénarios d'attaques définis au préalable. Un IDS par signature se compose :

- d'une ou plusieurs sondes, générant un flux d'événements, qui peuvent être de type réseau ou hôte ;

- d'une base de signatures ;
- d'un système de reconnaissance de motifs dans le flux d'événements.

La base de signatures Le taux de couverture de l'IDS dépend essentiellement de la qualité de la base de données puisque seules les attaques dont la signature est présente dans la base sont susceptibles d'être détectées. Les signatures sont décrites à l'aide de langages de description d'attaques [EVK02, CO00, MM01]. Elles sont la plupart du temps définies par un opérateur bien que des travaux récents permettent la génération automatique des signatures [BNS+06, NS05].

La base de signatures doit également être maintenue :

- Les nouvelles attaques détectées par la communauté doivent être intégrées à la base.
- Suivant les choix de l'administrateur de sécurité, les signatures qui ne correspondent plus à une possible intrusion (parce qu'un logiciel ou un système d'exploitation a été mis à jour, remplacé ou supprimé par exemple) peuvent être enlevées de la base.

La maintenance de la base de signatures est une tâche importante. Sans maintenance, l'IDS ne peut détecter les nouvelles attaques. La rapidité de la propagation de certains vers comme Slammer [MPS+03] montre également une limite de cette approche car le temps de mise à jour de la base de signatures par l'administrateur est supérieur au temps de propagation du ver.

Le système de reconnaissance de motifs Le système de reconnaissance de motifs est chargé d'identifier les motifs présents dans la base de signature, dans le flux d'événements. Différents systèmes de reconnaissance de motifs ont été définis dans la littérature. Cela va de systèmes simples à base de règles comme [Pax98] ou de correspondances de chaînes de caractères [MHL94] (*string matching*) à des systèmes bien plus complexes à base de systèmes experts comme [PN97] ou de modélisation d'états [EVK02] qui peuvent apporter suffisamment d'abstraction pour détecter des attaques inconnues mais qui font partie d'une même classe d'attaques. On pourra consulter la classification d'Axelsson [Axe00b] pour plus de détails sur ces systèmes.

Discussion Généralement, la pertinence des IDS suivant cette approche est bonne car les signatures correspondent à des éléments caractéristiques des attaques qui ne sont pas présents dans le flux d'événements en absence d'attaques. Cependant, les signatures sont souvent généralisées pour que l'IDS soit capable de détecter les modifications des attaques. Cette généralisation des signatures peut entraîner une baisse de la pertinence de l'IDS.

Les signatures sont, de manière générale, associées à des attaques spécifiques. En cas d'alertes, il est alors assez facile d'identifier exactement l'attaque voire la vulnérabilité qui a pu être exploitée ou non suivant la réussite de l'attaque. L'administrateur a alors la possibilité d'utiliser les informations présentes dans

des bases de vulnérabilités pour prendre les mesures adéquates, notamment à la remise du système dans un état sûr.

Les IDS suivant cette approche sont généralement performants d'un point de vue utilisation de ressources. En particulier pour les IDS au niveau réseau, l'IDS ne fonctionne pas sur les machines rendant un service dans le système. Les performances des services rendus par le système ne sont pas influencées significativement par la présence de l'IDS.

Les signatures décrivent généralement des attaques qui peuvent réussir ou non. Très peu d'IDS par signature sont capables de faire la différence entre une attaque ayant échoué et une intrusion. La présence d'attaques peut être une information importante pour l'administrateur de sécurité ; celle d'intrusions est cependant plus importante.

Les deux principaux avantages de solutions suivant l'approche par signature sont la pertinence des alertes et une certaine facilité de mise en place de l'IDS. Cependant, cette approche présente certaines limites dont la plus gênante est de ne pouvoir détecter que les attaques connues.

1.1.1.2 Détection d'intrusions comportementale

La détection d'intrusions comportementale a été la première approche proposée et développée. Anderson [And80] propose de détecter des violations de la politique de sécurité du système en observant le comportement des utilisateurs et en le comparant à un modèle du comportement considéré comme normal, appelé *profil*.

D'une manière générale, l'approche comportementale comporte deux phases : une phase d'apprentissage où le profil est constitué en observant le comportement de l'entité surveillée et une phase de détection pendant laquelle l'IDS observe le comportement de l'entité, mesure la similarité entre ce dernier et le profil et émet une alerte si la déviation est trop importante.

L'idée principale de cette approche est de considérer toute déviation, toute anomalie dans le comportement comme une intrusion. Cette hypothèse est certainement fausse : des événements ou des comportements rares peuvent tout à fait être légitimes du point de vue de la politique de sécurité du système. Le système est susceptible d'émettre des faux positifs. Tant que le nombre de faux positifs reste suffisamment faible, la méthode peut être valide.

Cela conduit à poser deux questions essentielles, dans le domaine de la détection d'intrusions comportementale, sur le caractère correct et complet du modèle de comportement normal (voir la figure 1.1).

Le modèle de comportement normal est dit correct s'il ne modélise que le comportement légitime, du point de vue de la politique de sécurité, de l'entité surveillée. Toutes les intrusions sont alors détectées par l'IDS : il n'y a pas de faux négatif.

Le modèle de comportement normal est dit complet s'il modélise entièrement le comportement légitime, du point de vue de la politique de sécurité, de l'entité surveillée. Dans ce cas, toutes les alertes correspondent à des intrusions : il n'y a pas de faux positifs.

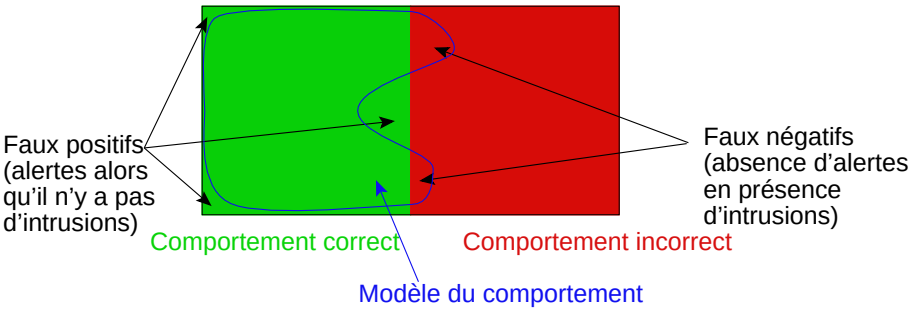


FIGURE 1.1 – Caractères complet et correct du modèle de comportement normal

L'idéal est d'obtenir un modèle à la fois complet et correct. Le choix de la méthode de modélisation, des attributs à considérer dans les événements observés vont avoir un impact important sur le caractère correct et complet du modèle.

Plusieurs méthodes de modélisation ont été proposées pour établir le profil de l'entité surveillée : modèles statistiques [LTG+90, ALJ+95], systèmes experts [VL89], réseaux de neurones [DBS92], approche immunologique [DFH96].

Modèles statistiques Dans cette approche, le profil est établi en observant la valeur de certains paramètres du système considéré comme des variables aléatoires. Pour chaque paramètre du système, un modèle statistique est utilisé pour établir la distribution de la variable aléatoire correspondante. Une fois le modèle établi, un vecteur distance est calculé entre le flux d'événements observés et le profil. Si la distance dépasse un certain seuil, une alerte est émise.

Les premiers IDS utilisant un modèle statistique comme, par exemple, celui proposé par Anderson [And80] ou IDES [LTG+90] (*Intrusion Detection Expert System*) ciblent la détection de comportements anormaux d'utilisateurs en étudiant des paramètres comme le temps processeur, la durée des sessions, le nombre de tentatives de *login*, etc. NIDES [ALJ+95] (*Next Intrusion Detection Expert System*) propose des améliorations par rapport à IDES en combinant les approches comportementales et par signature et étend cette approche à la modélisation statistique d'applications.

Systèmes experts Dans cette approche, le profil est établi en observant le flux d'événements pour en déduire un certain nombre de règles qui décrivent le comportement normal du système. Pendant la phase de détection, le système applique les règles au flux d'événements et vérifie si ce flux d'événements respecte ou non les règles apprises.

Ce modèle est utilisé dans le projet W&S [VL89] (*Wisdom and Sense*) pour détecter des comportements inhabituels chez les utilisateurs d'un système d'information.

Réseaux de neurones Un réseau de neurones peut être utilisé pour modéliser le comportement du système et apprendre à classifier les comportements normaux et anormaux. Ce modèle permet de prendre naturellement en compte des séries temporelles soit avec des réseaux de neurones récurrents ou des techniques de fenêtre glissante en entrée du réseau de neurones.

C'est la méthode employée dans [DBS92] pour modéliser le comportement des utilisateurs d'un système informatique. Dans ce travail, le réseau de neurones est utilisé en conjonction avec un système expert pour décider de la présence d'intrusions ou non.

Approche Immunologique Cette méthode a été proposée par Forrest et al. [FHSL96] et vise à détecter les comportements anormaux d'applications en observant les séquences d'appels système qu'effectue l'application surveillée. Pendant la phase d'apprentissage, toutes les séquences d'appels système d'une taille donnée sont stockées dans une base de séquences et constitue le profil. Lors de la phase de détection, une alerte est émise lorsqu'une séquence d'appels système effectués par l'application n'est pas présente dans la base de séquences.

Discussion L'avantage majeur de l'approche comportementale par rapport à l'approche par signature est de ne pas chercher à caractériser les intrusions mais le comportement attendu du système et donc de pouvoir détecter des intrusions inconnues.

De manière générale, les IDS fondés sur cette approche sont fiables car une intrusion génère souvent une anomalie dans le comportement observé. Cela reste cependant une question ouverte, comme le font remarquer Myers [Mye80] et Anderson [And80] et comme le montrent les développements récents dans le domaine des *mimicry attacks* [TKM02, WS02].

Par contre, ces IDS sont, généralement, peu pertinents. Il y a relativement peu d'études de performances des IDS comportementaux en termes de faux positifs : Helman et Liepins [HL93] étudient les performances à la fois théoriques des modèles statistiques et pratiques d'un détecteur statistique simple et de W&S [VL89]. Cette étude montre que les résultats de ces IDS restent loin des objectifs fixés par la DARPA : taux de détection de 99% pour un taux de faux positifs inférieurs 0,1% [McH01].

De plus, la phase d'apprentissage présente quelques problèmes : pour la plupart des méthodes de modélisation, il faut s'assurer que la base d'apprentissage soit exempte d'intrusions. Dans le cas contraire, l'IDS risquerait d'apprendre des comportements intrusifs et ne serait donc pas capable de les détecter ensuite. Le comportement de l'entité surveillée peut également évoluer au cours du temps, il est possible de modifier le profil continûment pendant la phase de détection pour que ce dernier représente toujours le plus fidèlement possible le comportement de l'entité. Dans ce cas, le système peut apprendre progressivement des comportements intrusifs introduits par un attaquant.

Dans cette thèse, nous avons choisi une approche différente de celles proposées dans les travaux en détection d'intrusions comportementale en décidant de ne pas construire le modèle de comportement normal explicite mais en utilisant plusieurs composants logiciels en parallèle et en comparant leur comportement. Le comportement de chaque logiciel est considéré comme un modèle de comportement normal pour les autres logiciels : on parle ainsi de modèle implicite. Ce modèle est, de manière évidente, incorrect car les composants logiciels contiennent des vulnérabilités. Pour limiter le nombre de faux positif, il faut que ce modèle soit le plus complet possible.

Ce modèle est également incomplet, comme le montre la présence de faux positifs dans le cadre de nos expérimentations.

Pour tester les méthodes de détection d'intrusions que nous avons proposées, nous avons choisi de développer un IDS pour serveurs web. Dans la sous-section suivante, nous présentons les travaux précédents plus spécifiques à la détection d'intrusions pour serveurs web quelque soit leur approche.

1.1.2 Détection d'intrusions web

Les serveurs web sont un environnement de test intéressant pour la détection d'intrusions, d'une part, par leur importance et par l'universalité du protocole HTTP [FGM+99] (*Hypertext Transfert Protocol*) et, d'autre part, par le nombre de vulnérabilités les frappant.

Les serveurs web sont la vitrine des entreprises, associations, états, voire des individus par l'intermédiaire des blogs sur l'Internet. Ils sont, dans certains cas, une source de revenus importants (commerce en ligne par exemple). De plus en plus d'applications web sont déployées sur l'Internet : applications médicales, commerce en ligne, bureaux virtuels, services de cartographie, réseaux sociaux, services de paiement, services administratifs (paiement des impôts notamment), etc.

Ces serveurs et les applications fonctionnant sur ceux-ci sont accessibles de l'extérieur et peuvent présenter des vulnérabilités. Les serveurs présentent beaucoup moins de vulnérabilités qu'il y a quelques années, les développeurs ayant pris conscience de l'importance de la sécurité. Ce n'est, par contre, pas encore le cas pour les applications web : d'après Robertson et al. [RVKK06], 25% des entrées CVE (*Common Vulnerabilities and Exposures*) de 1999 à 2005 sont liées à des vulnérabilités web ; de plus, ce chiffre ne tient pas compte de toutes les applications développées en interne, dans différentes entreprises, pour répondre à des besoins particuliers. Les pirates informatiques essaient de profiter de ces vulnérabilités pour y installer de faux sites dans le but de faire de l'hameçonnage (*phishing*) ou pour y installer des logiciels malveillants (*malwares*) qui infecteront les clients visitant le site [PMRM08].

La sécurité des serveurs web et des applications fonctionnant sur ces serveurs est une priorité tant pour l'entité représentée par le serveur que pour les clients visitant ces serveurs.

Les outils de détection d'intrusions « génériques » peuvent être utilisés pour détecter les intrusions contre les serveurs web : les NIDS comme Bro [Pax98], NSM [HDL+90, MHL94] (*Network Security Monitor*) ou Snort [Roe99] ou les *Host-based* IDS qui surveillent le comportement de programmes tels que ceux développés par Forrest et al. [FHSL96, WFP99] ou Ghosh [GMS00] par exemple. Bien que ces IDS n'aient pas été spécialement évalués dans le domaine de la détection d'intrusions web, le web reste un domaine de prédilection pour la détection d'intrusions notamment pour les IDS au niveau réseau : dans la version 2.3.3 de Snort, 1064 signatures sur les 3111 sont consacrées à la détection des attaques web.

Les outils de détection d'intrusions utilisés pour détecter les attaques contre les serveurs web utilisent principalement une approche par scénario bien que des approches comportementales soient apparues récemment. Nous allons donc présenter les différents IDS spécifiques au web suivant leur approche de détection.

1.1.2.1 Les approches par signature

Les IDS par signature spécifiques au web sont pour la plupart des HIDS au niveau applicatif. Ils évitent certains écueils des NIDS : reconstruction des paquets, perte de paquets en cas de charge, vulnérabilité aux techniques d'évasion, gestion de la cryptographie, etc. La plupart de ces outils utilisent les fichiers d'audit des serveurs web comme source d'événements.

WWWstat [Fie96] est principalement un programme destiné à établir des statistiques sur l'utilisation du serveur web. Il ne réalise pas de détection d'intrusions par lui-même mais sa sortie peut être utilisée à des fins de détection d'intrusions manuelle.

Autobuse [Tay99] est un cadre d'applications pour l'analyse de fichiers d'audit en provenance de pare-feu ou de serveurs web. Il recherche parmi les différentes entrées de ces fichiers la présence d'attaques connues et alerte l'administrateur (notamment par mail).

Almgren et al. [ADD00] ont proposé également un système analysant les logs générés par le serveur web, recherchant des motifs qui correspondent à des attaques connues. L'analyse peut se réaliser en temps réel et n'affecte pas les performances du serveur web. Ce système est également capable d'apprendre de nouvelles attaques en surveillant plus particulièrement les activités des clients considérés comme suspects. Cependant les attaques analysées sont limitées à l'exploitation de vulnérabilités dans les scripts CGI. Les attaques concernant le serveur en lui-même, comme certains dépassements de tampon, ne sont pas détectées. De plus, même si un module combinant les alertes est fourni, les attaques complexes nécessitant plusieurs étapes ne peuvent être modélisées et ne sont donc pas détectées.

Un autre système WebSTAT [VRKK03] fondé sur le cadre d'applications STAT [VVK03] utilise un langage de haut-niveau STATL [EVK02] pour décrire les attaques en termes d'états et de transitions. Ce langage permet la description d'attaques complexes. La détection d'intrusions se fait en temps réel et plusieurs

autres sources d'événements peuvent être utilisées (même si cela n'a pas encore été fait) comme les fichiers d'audit du système d'exploitation ou encore des flux réseaux. Cette méthode présente une dégradation en performance et les auteurs n'ont pas présenté de résultats concernant la fiabilité et la pertinence du système de détection.

Les approches précédentes utilisent comme entrée les fichiers de log du serveur web. Ces outils sont donc vulnérables aux attaques capables d'empêcher le serveur d'écrire dans le fichier d'audit ou aux attaques compromettant l'intégrité du fichier d'audit. De plus, les fichiers d'audit des serveurs ne contiennent qu'une partie de la requête HTTP (la première ligne essentiellement). Des éléments caractéristiques des intrusions peuvent cependant se trouver dans les autres parties de la requête. D'autres outils de détection ont été proposés pour prendre en compte ce problème.

Almgren et Lundqvist [AL01] ont proposé un système de détection d'intrusions intégré à un serveur Apache. L'avantage de cette solution réside dans sa capacité à détecter les intrusions à différents stades du traitement de la requête. Cette méthode entraîne également une dégradation dans les performances du serveur mais est également spécifique au serveur Apache. Cet outil a également la possibilité de recevoir des informations provenant d'autres sources notamment des flux réseaux.

L'IDS présenté ci-dessus est à rapprocher de ModSecurity [Ris08] qui est un module pour Apache permettant d'écrire des règles pour détecter, bloquer, modifier les requêtes parvenant au serveur Apache. ModSecurity peut également être utilisé en tant que *reverse proxy* ce qui permet notamment de faire du *virtual patching*, c'est-à-dire d'empêcher une vulnérabilité d'être exploitée dans une application web le temps que le correctif soit disponible pour cette application.

1.1.2.2 Les approches comportementales

Bien que les approches par signature soient effectives, elles posent certains problèmes. La plupart des applications web sont spécifiques et sont développées rapidement sans souci de sécurité particulier. Il est difficile d'écrire des signatures pour ces applications car il n'y a pas forcément de caractéristiques communes contrairement aux *buffer overflows*. Les entreprises n'ont pas forcément le temps et les ressources nécessaires pour employer un expert pour écrire ces signatures. L'approche comportementale semble ici adaptée à la nature des vulnérabilités.

Breach Security propose un produit nommé *WebDefend* [Sec08] qui modélise le trafic normal à destination et en provenance des applications web protégées et est ensuite capable de bloquer les attaques.

Kruegel et al [KV03, KVR05] ont proposé le premier IDS comportemental spécifique aux serveurs web. Il utilise les fichiers d'audit comme source d'événements. Il ne traite cependant que les requêtes de type GET comportant des

paramètres et se concentre donc sur la détection d'attaques contre les scripts CGI. Ce système utilise une approche statistique et étudie plusieurs caractéristiques des requêtes comme la longueur des valeurs des attributs, la distribution des caractères présents dans les valeurs des attributs, la présence ou l'absence d'attributs, l'ordre des attributs et la détermination si oui ou non la valeur d'un attribut fait partie d'un ensemble fini. Ce système nécessite donc une phase d'apprentissage, l'apprentissage se faisant différemment pour chaque caractéristique prise en compte. L'évaluation s'est faite sur chacune des caractéristiques étudiées indépendamment des autres et chaque caractéristique permet de détecter plus ou moins bien les attaques suivant leur type. Ce système repose également sur les fichiers d'audit du serveur web donc les attaques qui ciblent spécifiquement le serveur web ne seront pas détectées si le serveur web n'écrit pas dans son fichier d'audit ou si l'intégrité du fichier de log est compromise. Il ne peut pas détecter non plus les intrusions dont aucun élément caractéristique ne se trouve dans la première ligne de la requête HTTP. Les auteurs notent qu'il est possible d'intégrer leur IDS au serveur pour pouvoir traiter les requêtes entièrement comme cela est fait dans [AL01].

Robertson et al. [RVKK06] présentent une amélioration des travaux précédents en ajoutant à la détection d'anomalies deux composants : un composant permettant la génération de signatures d'alertes et groupant les alertes suivant les signatures et un composant permettant d'identifier les anomalies suivant des heuristiques. Le premier composant permet de réduire le nombre d'alertes présentées à l'administrateur en les groupant suivant leurs caractéristiques (distribution de caractères similaire, longueur similaire, etc). En suivant des heuristiques, le second associe les alertes à des groupes d'attaques spécifiques au web : remontée dans l'arborescence (*directory traversal*), injection SQL, *buffer overflow* et *cross-site scripting*.

Valeur et al. [VVKK06] proposent également une amélioration des travaux de Kruegel et Vigna [KV03]. Leur approche permet de limiter l'influence des faux positifs en intégrant les travaux de Kruegel et Vigna dans un reverse proxy qui redirige les requêtes HTTP anormales vers un serveur web ayant un accès plus restreint aux parties critiques du site.

Wang et al. [WS04] présentent un travail proche des travaux de Kruegel et al. [KTK02]. Ce travail consiste en la modélisation statistique de la distribution de la valeur des octets pour chaque paquet ou pour chaque connexion. Ils testent leur approche sur les données de DARPA 99 [LHF+00] et sur des données provenant d'un serveur universitaire. Ils appliquent leur approche notamment sur le port 80 dédié au trafic web et obtiennent des résultats corrects en termes de faux positifs.

Estévez-Tapiador et al. [ETGTDV04] modélisent les requêtes HTTP grâce à des chaînes de Markov. Ils proposent deux approches : l'une utilisant une extension des histogrammes de distribution de chaque caractère et l'autre prenant en compte des éléments de spécification du protocole HTTP pour effectuer un découpage en blocs syntaxiques. Les tests ont été effectués sur les données de DARPA 99 [LHF+00] pour le trafic normal et une base d'attaques de variantes de 86 attaques HTTP présentes dans la base ArachNIDS [Ara03]. La seconde

méthode donne des résultats bien meilleurs (d'un ordre de magnitude) en termes de faux positifs pour un taux de détection de 95%.

Ingham et al. [ISBF07] et Ingham [Ing07] proposent également une méthode portant sur toute la requête HTTP. Après l'application de certaines heuristiques, ils modélisent les requêtes HTTP grâce à des automates à états finis déterministes. Ils présentent un nouvel algorithme de généralisation pour définir l'automate à états finis et utilisent une technique similaire à celle de Robertson et al. [RVKK06] pour grouper les alertes et réduire le nombre d'alertes présentées à l'administrateur de sécurité. L'apprentissage de l'automate peut se faire même en présence d'attaques dans la base d'apprentissage, ce qui est un avantage par rapport aux autres méthodes proposées.

Une comparaison des techniques précédentes est présentée par Ingham et Inoue [II07]. Ils ont constitué une base de 63 attaques web et ont collecté un ensemble de requêtes HTTP pour 4 sites distincts. Ils proposent et mettent à disposition de la communauté un *framework* permettant la comparaison des outils de détection d'intrusions spécifiques au web. Ils montrent que les méthodes s'attachant aux unités lexicales (*token*) sont sensiblement plus performantes que les méthodes s'attachant à modéliser les requêtes au niveau des octets.

Nous ne pouvons malheureusement utiliser tel quel le *framework* qu'ils proposent. En effet, nous avons non seulement besoin des requêtes pour tester notre approche mais nous devons réellement les jouer sur les serveurs de notre architecture et observer leurs comportements. Il nous est donc nécessaire d'avoir, en plus des requêtes, les applications fonctionnant sur les serveurs et les données présentes sur les serveurs. Il nous est tout de même possible d'utiliser la base d'attaques en installant les différentes applications et serveurs nécessaire pour la réussite des exploits. Cela n'a pu malheureusement être fait, faute de temps mais est considéré dans la poursuite des travaux.

1.1.2.3 Une approche hybride

Une approche hybride a été proposée par Tombini et al. [TDMD04, DT05]. Cette approche consiste en la sérialisation d'un IDS comportemental suivi d'un IDS par signature. L'IDS comportemental permet de filtrer les requêtes normales et ainsi seules les requêtes détectées comme anormales sont passées à l'IDS par signature. Bien que l'IDS comportemental utilisé soit simple, ceci permet de réduire le nombre de faux positifs générés globalement. La source d'entrées est le fichier d'audit du serveur web. Cet IDS est donc soumis aux mêmes problèmes que les autres utilisant cette source de données.

1.1.2.4 Discussion

Les approches par signature montrent des limites quant à la détection des intrusions/attaques de par le fait que la plupart des vulnérabilités web sont spécifiques à des applications précises éventuellement développées en interne par les entreprises. Les méthodes comportementales sont donc une approche intéressante dans ce domaine. Les premiers IDS comportementaux [KV03, TDMD04]

proposés ne prenaient en compte que la première ligne de la requête HTTP, la seule présente dans les fichiers d'audit des serveurs et ne sont capables de détecter que les attaques qui vont influer sur cette partie de la requête. Des approches plus récentes prennent en compte la sémantique des requêtes en effectuant une analyse lexicale du protocole [ETGTDV04, ISBF07]. Ces outils semblent obtenir de meilleurs résultats [II07]. Notre approche se distingue par la prise en compte du comportement des serveurs web et non seulement des caractéristiques de la requête et nous cherchons à détecter les anomalies dans le comportement des serveurs pour identifier les intrusions, ceci dans le but de différencier les intrusions des éventuelles attaques ou requêtes anormales.

Nous présentons dans la suite un panorama rapide du domaine de la sûreté de fonctionnement, plus précisément des concepts et méthodes qui concernent les travaux de cette thèse : modèle faute - erreur - défaillance, tolérance aux fautes, diversification fonctionnelle.

1.2 Détection d'erreurs

Notre approche de détection d'intrusions est fondée sur une technique du domaine de la sûreté de fonctionnement : la diversification fonctionnelle. Dans cette section, nous définissons les concepts fondamentaux de la sûreté de fonctionnement et les divers moyens d'assurer les propriétés de sûreté informatique. La fin de cette section se consacre à l'analyse des concepts du domaine de la sécurité dans le domaine de la sûreté de fonctionnement, travaux réalisés dans le cadre du projet européen MAFTIA [AAC+03].

1.2.1 Le modèle faute - erreur - défaillance

Les définitions sont issues du guide de la sûreté de fonctionnement [ABC+95]. La sûreté de fonctionnement d'un système informatique est la propriété qui permet à ses utilisateurs de placer une confiance justifiée dans le service qu'il leur délivre. Un utilisateur n'est pas forcément un être humain mais peut tout à fait être un autre système qui interagit avec le service considéré. Le service délivré par le système est le comportement perçu par ses utilisateurs. La sûreté d'un système peut être analysée suivant différentes propriétés appelées attributs :

- le fait d'être prêt à l'utilisation conduit à la *disponibilité* ;
- la continuité du service conduit à la *fiabilité* ;
- la non-occurrence de conséquences catastrophiques pour l'environnement conduit à la *sécurité-innocuité* ;
- la non-occurrence de divulgations non-autorisées de l'information conduit à la *confidentialité* ;
- la non-occurrence d'altérations inappropriées de l'information conduit à *l'intégrité* ;
- l'aptitude aux réparations et aux évolutions conduit à la *maintenabilité*.

L'association, à la confidentialité, de l'intégrité et de la disponibilité vis-à-vis des actions autorisées, conduit à la *sécurité-confidentialité*. La sécurité telle que nous l'entendons, c'est-à-dire la sécurité-confidentialité dans le cadre de la sûreté de fonctionnement, fait donc partie intégrante du domaine de la sûreté de fonctionnement.

Dans un système donné, on considère que ces propriétés peuvent être mises en défaut par des entraves au sein du système. Ces entraves sont de trois types :

- les *défaillances* qui surviennent lorsque le service délivré dévie de l'accomplissement de la fonction du système ;
- les *erreurs* sont les parties de l'état du système susceptibles d'entraîner des défaillances ;
- les *fautes* sont les causes adjugées ou supposées des erreurs.

Ces entraves forment une chaîne causale logique : une faute peut entraîner une erreur, qui peut elle-même entraîner une défaillance. De plus cette chaîne est récursive, et une défaillance externe à un composant, peut provoquer une faute interne dans un autre composant, qui peut elle-même entraîner une erreur, etc.

Les fautes peuvent être classées suivant 5 critères : leur cause phénoménologique, leur nature, leur phase de création ou d'occurrence, leur situation par rapport aux frontières du système, et leur persistance.

On distingue suivant leur cause phénoménologique : les fautes physiques et les fautes dues à l'homme. Suivant la nature des fautes, on distingue : les fautes accidentelles et les fautes intentionnelles. Suivant leur phase de création ou d'occurrence, les fautes de développement et les fautes opérationnelles. Suivant la situation des fautes par rapport aux frontières du système : les fautes internes et les fautes externes. Suivant leur persistance : les fautes permanentes et les fautes temporaires.

Bien que l'on puisse détecter des fautes physiques grâce à la diversification fonctionnelle, dans le cadre de nos travaux, nous nous intéressons aux fautes dues à l'homme. Celles-ci peuvent être réparties en quatre classes de fautes combinées :

- les fautes de conception, qui sont des fautes de développement accidentelles ou intentionnelles sans volonté de nuire ;
- les fautes d'interaction, qui sont des fautes externes, accidentelles ou intentionnelles sans volonté de nuire ;
- les logiques malignes (ver, virus, bombe logique, porte dérobée, cheval de Troie) qui sont des fautes internes intentionnellement nuisibles ;
- les intrusions qui sont des fautes opérationnelles externes intentionnellement nuisibles.

Une faute est active lorsqu'elle produit une erreur. Une faute active est soit une faute interne qui était préalablement dormante et qui a été activée par le processus de traitement, soit une faute externe. Une faute interne peut cycler entre ses états dormant et actif.

Une erreur peut être latente ou détectée ; une erreur est latente tant qu'elle n'a pas été reconnue en tant que telle ; une erreur est détectée par un algorithme

ou un mécanisme de détection. Une erreur peut disparaître sans être détectée. Par propagation, une erreur crée de nouvelles erreurs.

Une défaillance survient lorsque, par propagation, une erreur affecte le service délivré par le système. Cette défaillance peut alors apparaître comme une faute du point de vue d'un autre composant. On obtient ainsi la chaîne fondamentale suivante :

$$\ldots \rightarrow \text{défaillance} \rightarrow \text{faute} \rightarrow \text{erreur} \rightarrow \text{défaillance} \rightarrow \text{faute} \rightarrow \ldots$$

Les flèches dans cette chaîne expriment la relation de causalité entre fautes, erreurs et défaillances. Elles ne doivent pas être interprétées au sens strict : par propagation plusieurs erreurs peuvent être créées avant qu'une défaillance ne survienne.

Les moyens, permettant d'assurer que les attributs de la sûreté de fonctionnement sont présents et maintenus au sein du système, sont classés selon quatre axes :

- prévention des fautes : comment empêcher l'occurrence ou l'introduction de fautes ;
- tolérance aux fautes : comment fournir un service à même de remplir la fonction du système en présence de fautes ;
- élimination des fautes : comment réduire la présence des fautes ;
- prévision des fautes : comment estimer la présence, la création et les conséquences des fautes.

Ces moyens sont complémentaires et dépendants et doivent être utilisés de manière combinée. Nous allons nous intéresser dans la suite de cette section aux divers techniques de tolérance aux fautes et surtout à une technique particulière : la diversification fonctionnelle.

1.2.2 La tolérance aux fautes

La tolérance aux fautes [Avi67] est mise en œuvre par le traitement des erreurs et par le traitement des fautes [AL81]. Le traitement d'erreur est destiné à éliminer les erreurs, de préférence avant qu'une défaillance ne survienne. Le traitement de faute est destiné à éviter qu'une ou des fautes ne soient activées de nouveau.

Le traitement d'erreur fait appel à trois types de primitives :

- la détection d'erreur permet d'identifier un état erroné ;
- le diagnostic d'erreur permet d'estimer les dommages créés par l'erreur qui a été détectée et par les erreurs éventuellement propagées avant la détection ;
- le recouvrement d'erreur permet de substituer un état exempt d'erreur à un état erroné, cette substitution pouvant prendre trois formes :

  la reprise qui substitue à l'état courant un état préalablement sauvegardé au niveau d'un point de reprise ;

la poursuite qui trouve un état à partir duquel le système peut fonctionner ;

la compensation d'erreur qui construit un état exempt d'erreur pour peu qu'un niveau de redondance suffisant ait été introduit dans l'application.

Le traitement des fautes, de son côté, peut être divisé en trois étapes : le diagnostic de fautes qui consiste à déterminer les causes des erreurs, la passivation des fautes qui permet d'empêcher une nouvelle activation des fautes, et la reconfiguration qui vise à modifier l'état du système pour qu'il puisse continuer à délivrer un service, même dégradé.

La compensation d'erreur peut être appliquée systématiquement, même en l'absence d'erreur, procurant alors un masquage de faute (par exemple par vote majoritaire). La détection d'erreur n'est pas alors stricto sensu nécessaire pour effectuer le recouvrement ; cependant, afin d'éviter une diminution non perçue de la redondance disponible lors de la défaillance d'un composant, les mises en œuvre pratiques du masquage comportent généralement une détection d'erreur, qui peut dans ce cas être effectuée après le recouvrement.

Nos travaux entrent dans le cadre de la détection d'erreur et de la compensation d'erreur. Nous cherchons à détecter des états du système où une violation de la politique de sécurité a eu lieu et nous cherchons à masquer cet état de l'extérieur par la redondance des composants.

1.2.3 La diversification fonctionnelle

Il existe plusieurs approches de tolérance aux fautes de conception. La programmation défensive cherche à éviter qu'une défaillance d'un composant n'entraîne la défaillance de tout le système. La diversification fonctionnelle cherche, quant à elle, à assurer la continuité du service. Elle suppose l'existence de plusieurs composants capables d'assurer la même tâche et conçus et réalisés de manière différente à partir de la même spécification. La diversification fonctionnelle est une approche dédiée au masquage de faute mais permet également de détecter des erreurs.

Il existe trois techniques de base de tolérance aux fautes par diversification fonctionnelle : les blocs de recouvrement, la programmation N-autotestable et la programmation N-versions. Nous allons décrire brièvement les deux premières dans la suite de cette sous-section et présenter la programmation N-versions de manière plus complète dans la sous-section suivante. La technique de détection que nous proposons est en effet fondée sur la programmation N-versions, les deux autres techniques ne pouvant être appliquées soit par définition pour les blocs de recouvrement soit à cause d'une trop grande complexité de mise en œuvre pour la programmation N-autotestable.

Toutes les techniques de diversification fonctionnelle nécessitent la présence de plusieurs variantes c'est-à-dire d'exemplaires différents d'un produit. Ces techniques nécessitent également la présence d'un décideur, destiné à fournir un résultat supposé exempt d'erreur à partir des exécutions des variantes. La spécification commune aux variantes doit mentionner explicitement les points de décision, c'est-à-dire quand les décisions doivent être prises et également les

données sur lesquelles les décisions doivent être prises.

La notion de diversification du logiciel pour la tolérance aux fautes de conception a été formulée dans les années 1970 [Elm72, Ran75, CA78] et a été utilisée ensuite en pratique dans les systèmes critiques (notamment dans le domaine aéronautique) pour assurer des propriétés de sécurité-innocuité.

Le gain en fiabilité est obtenu car l'impact des fautes indépendantes entre les différentes variantes (qui constituent la majorité des cas) devient négligeable. Le gain est limité par les fautes corrélées, résultant soit d'une spécification erronée (commune à toutes les variantes) soit de dépendances dans les conceptions et réalisations séparées des variantes. De surcroît, il a été constaté que l'exécution de logiciels multi-variantes peut mettre en évidence des fautes de spécification.

1.2.3.1 Les blocs de recouvrement

Dans le cadre des blocs de recouvrement [Ran75], les variantes sont appelées des alternants et le décideur est un test d'acceptation, qui est appliqué séquentiellement aux résultats fournis par les alternants : si les résultats fournis par l'alternant primaire ne satisfont pas le test d'acceptation, l'alternant secondaire est exécuté, et ainsi de suite jusqu'à la satisfaction du test d'acceptation ou l'épuisement des alternants disponibles. Dans ce dernier cas, le bloc de recouvrement est globalement défaillant.

1.2.3.2 La programmation N-autotestable

La programmation N-autotestable [LABK90] nécessite l'exécution parallèle d'au moins deux composants autotestables, un composant autotestable étant constitué soit de l'association d'une variante et d'un test d'acceptation, soit de deux variantes et d'un algorithme de comparaison. Chaque composant fournit donc un résultat acceptable ou non. Si tous les composants du système ne fournissent pas de résultat acceptable, le logiciel est alors défaillant. Ici, l'exécution des variantes se fait donc de manière parallèle et non séquentielle comme dans le cas des blocs de recouvrement.

1.2.4 La programmation N-versions

Dans la programmation N-versions [AC77, AK84, ALS88, Avi95], les variantes sont appelées des versions et le décideur effectue un vote sur les résultats de toutes les versions. Le schéma d'exécution des variantes est donc parallèle à l'instar de la programmation N-autotestable ; par contre, le jugement sur l'acceptabilité des résultats est relatif par rapport aux résultats produits par toutes les variantes ; ce n'est pas le cas dans les deux techniques décrites précédemment.

La programmation N-versions est définie comme la génération indépendante de N 2 programmes fonctionnellement indépendants à partir de la même spécification initiale. La spécification initiale doit également comporter les points

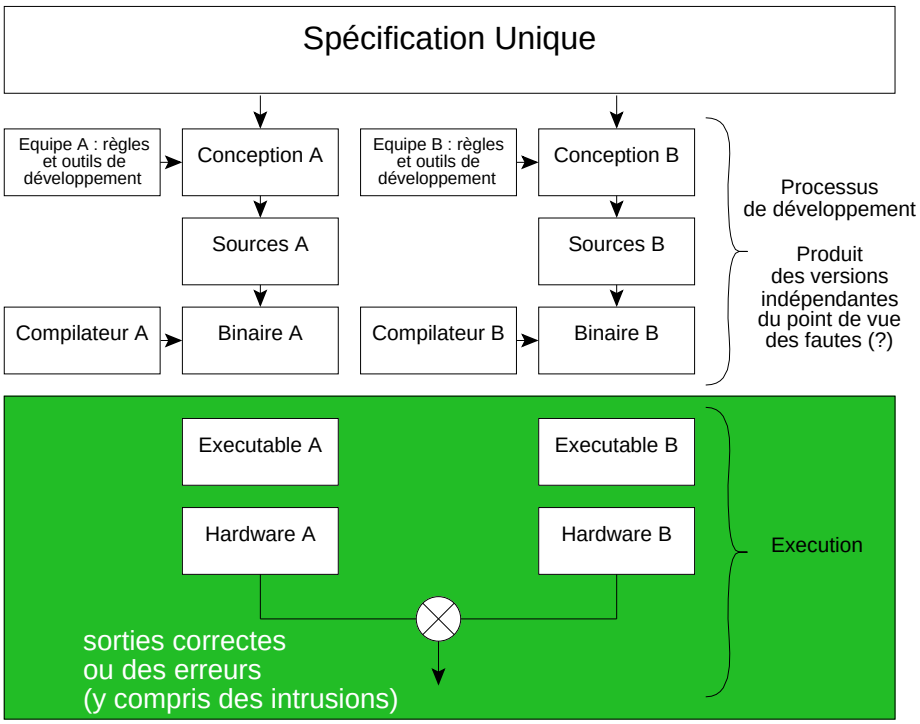


FIGURE 1.2 – Processus de développement des versions en programmation N-versions

de comparaison entre les versions (FIG. 1.2).

La programmation N-versions comporte trois éléments distincts :
- le processus de spécification initiale qui doit assurer l'indépendance et l'équivalence fonctionnelle des N versions (ce qui implique qu'en absence d'erreurs, les résultats doivent être identiques ou au moins comparables) ;
- la production des programmes qui doivent pouvoir s'exécuter en parallèle et présenter les valeurs demandées aux différents points de décision ;
- l'environnement qui assure l'exécution des N logiciels et fournit les algorithmes de décision aux points de décision.

L'objectif de la programmation N-versions est de maximiser l'indépendance entre le développement des versions et de minimiser ainsi la probabilité que deux ou plus des versions produisent des erreurs similaires qui coïncident temporellement. Le succès de la programmation N-versions repose donc sur la conjecture suivante : la génération indépendante des N versions réduit considérablement la probabilité que des fautes logicielles identiques soient produites dans deux ou plus des versions. Cela implique que la spécification initiale doit être exempte

d'erreurs et laisser la plus grande liberté possible aux développeurs quant au choix de réalisation.

On peut se poser la question de l'efficacité de la programmation N-versions. Des éléments de réponse sont apportés par trois types d'étude :

- des expériences de l'application de la programmation N-versions dans le monde de l'industrie ;
- des expériences dans des environnements contrôlés ;
- des modèles mathématiques des processus de défaillances de différentes versions.

Expériences industrielles De nombreuses expériences industrielles ont été réalisées notamment chez Boeing [WYF83] et Airbus [Tra88]. Cependant relativement peu de données ont été publiées. Aucun incident grave attribué à des fautes de conception/réalisation n'a été signalé concernant les logiciels critiques d'un point de vue de la sécurité-innocuité utilisant la programmation N-versions. Il semble raisonnable de penser que la diversification fonctionnelle a contribué à la sûreté de fonctionnement de ces applications mais il est difficile de savoir jusqu'à quel point. La programmation N-versions reste très utilisée dans l'industrie pour un certain nombre de projets critiques.

Expériences menées dans des environnements contrôlés Des expériences [KAU+86, ALS88] menées dans des environnements contrôlés ont également été réalisées notamment durant les années 1980 sous le sponsorship de la NASA. Ces études avaient pour but de vérifier statistiquement que des versions développées indépendamment conduisent à des fautes indépendantes. Ces études montrent cependant que des fautes peuvent être communes à plusieurs versions. Même si l'hypothèse d'indépendance semble fausse, l'utilisation de la programmation N-versions conduit à une amélioration sensible de la sûreté de fonctionnement en moyenne car, même si des fautes corrélées existent entre certaines des versions produites, l'utilisation de 3 variantes parmi celles développées dans ces études, réduit le taux d'erreurs du système N-versions. Des critiques ont reproché à ces études l'utilisation d'étudiants en tant que développeurs ainsi que le possible manque d'indépendance dans la réalisation des différentes versions.

Modèles mathématiques Différents modèles mathématiques ont été proposés [EL85, LM89]. Nous allons ici décrire brièvement le modèle de Eckhardt et Lee [EL85] (EL). Dans le modèle EL, il y a deux sources basiques d'incertitude : d'une part, le choix aléatoire d'une entrée dans l'espace des entrées et d'autre part, le choix aléatoire d'un programme parmi l'espace des programmes. Le premier choix se fait suivant une loi de probabilité sur l'espace des entrées, cette loi établissant un profil des entrées. Le second choix représente le fait que l'écriture d'un programme revient en fait à son choix dans l'ensemble des programmes possibles. Certains programmes ont clairement une probabilité nulle d'être "choisis" (par exemple ceux qui ne s'intéressent pas du tout au problème posé). D'autres

ont une probabilité non-nulle d'être choisis (par exemple les programmes corrects résolvant le problème posé ainsi que ceux qui contiennent des fautes). La variable clé du modèle est la fonction de difficulté $\theta(x)$, définie par la probabilité qu'un programme choisi aléatoirement (suivant la loi de probabilité de choix d'un programme dans l'ensemble des programmes pour le problème posé) donnera un résultat faux sur l'entrée x. C'est une définition intuitive de la notion de difficulté : plus une entrée est difficile, plus grande est la chance qu'un programme inconnu échoue sur cette entrée. Il est alors clair que la probabilité qu'un programme choisi aléatoirement échoue sur une entrée choisie aléatoirement est : $E_X(\theta(X))$. La probabilité que deux programmes $_1$ *et* $_2$, développés indépendamment, c'est-à-dire vérifiant $\forall x\ P(\ _1$ *échoue sur* $x \mid\ _2$ *échoue sur* $x) = P(\ _1$ *échoue sur* $x) = \theta(x)$ échouent sur une entrée choisie aléatoirement est alors : $\sum_x P(X = x)P(\ _1$ *et* $\ _2$ *échouent* $|x$ *est l'entrée*$) = E_X([\theta(X)]^2) = E_X(\theta(X)^2) + Var_X(\theta(X))$.

La variance $Var_X(\theta(X))$ est toujours positive donc la probabilité que deux programmes développés indépendamment échouent sur une entrée choisie aléatoirement est donc supérieure à la supposition "naïve" d'indépendance donnée par $E_X(\theta(X)^2)$. L'intérêt de ce modèle est de définir précisément la notion d'indépendance des processus de développement de deux versions et de la relier à l'indépendance statistique. Ce modèle reste cependant très théorique puisqu'il décrit des objets auxquels on ne peut avoir accès dans la pratique (loi de probabilité pour le choix d'un programme notamment).

Ces trois types d'études montrent que la programmation N-versions est une technique efficace dans le domaine de la sûreté de fonctionnement et surtout pour la sécurité-innocuité. Dans nos travaux, nous avons cependant choisi d'utiliser l'architecture de la programmation N-versions avec des COTS dans le but de détecter des intrusions.

1.2.5 La diversification de COTS

L'utilisation de COTS en lieu et place des versions spécialement développées comme c'est le cas en diversification fonctionnelle pose évidemment certains problèmes [ABB+00] que nous détaillons ci-après.

En effet, les techniques de diversification fonctionnelle supposent que les variantes soient conçues exactement à partir de la même spécification fonctionnelle. L'utilisation de COTS n'assure pas ce premier point. Par exemple, dans le cadre des serveurs HTTP, certains serveurs, comme thttpd, n'intègrent pas de support du langage php alors que c'est une caractéristique commune à d'autres serveurs web COTS : Apache, Lighttpd, IIS, etc. IIS supporte la technologie ASP ce qui n'est pas le cas d'Apache ou de Lighttpd.

La spécification doit mentionner explicitement les points de décision, c'est-à-dire les instants dans le traitement où le décideur doit être invoqué et les données sur lesquelles les décisions doivent être prises. Étant donné que les COTS n'ont pas été conçus dans l'optique d'être utilisés dans une architecture de diversification fonctionnelle, leur spécification ne comprend évidemment pas

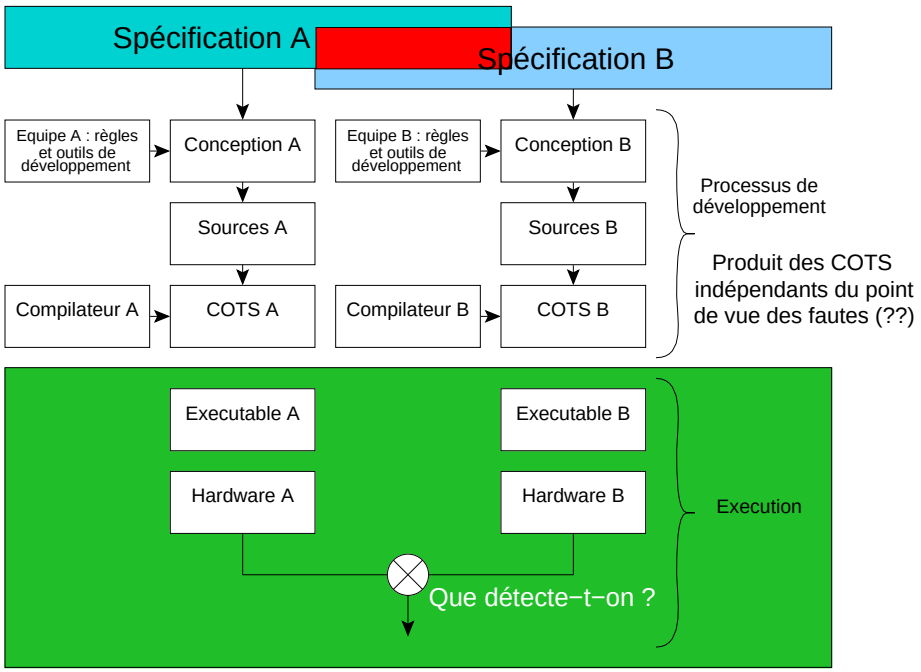


FIGURE 1.3 – Processus de développement et utilisation de COTS dans une architecture N-versions

ces informations.

Dans le cadre de la programmation N-versions, les versions sont développées de manière à limiter la probabilité de défaillances communes, ou, autrement dit, d'augmenter la probabilité que l'activation des fautes soient décorrélées. Les COTS, de par leur nature, ne sont pas développés avec les mêmes contraintes : les équipes de développement de différents COTS peuvent communiquer entre elles, dans le cadre du respect de la licence d'un COTS libre, il est possible de réutiliser le code de ce COTS dans un autre COTS, etc. Les COTS ne sont pas développés dans l'optique de limiter la probabilité de défaillances communes. Il est alors nécessaire de vérifier si les COTS que l'on souhaite utiliser vérifient bien cette propriété. Cela est partiellement possible soit en utilisant un jeu de tests, de préférence, le plus complet possible, soit en étudiant leurs fautes connues ou, plus précisément en ce qui concerne la détection d'intrusions, leurs vulnérabilités connues. La Figure 1.3 résume certains de ces points.

Il est possible d'envisager d'utiliser des COTS devant se conformer à une interface standardisée. Les invocations de l'interface standardisée peuvent constituer des points de décision. Ceci résout les deux premiers points évoqués ci-dessus. Il reste cependant un problème lié au non-déterminisme de l'exécution des COTS en l'absence d'activation des fautes. Il est possible que des appels iden-

tiques aux COTS conduisent à des comportements différents dus, par exemple, à l'ordonnancement de *threads* par le système d'exploitation que l'on ne peut pas forcément contrôler.

C'est cette approche que nous avons suivie dans la première méthode de détection d'intrusions que nous avons proposée dans cette thèse. Cela avait été envisagé auparavant dans le cadre d'une instance de l'architecture GUARDS destinée aux applications de propulsion nucléaire [PABD+99]. Cette approche a été étudiée plus récemment dans le cadre du projet DOTS [GPSS04, PS03] (*Diversity with Off-The-Shelf components*) pour des bases de données SQL.

Le projet DOTS s'est intéressé à la réalisation de *wrappers* de protection pour les COTS [vdMRSJ05] en introduisant le concept de *Acceptable Behaviour Constraints*. Gashir et al. [GPS04] ont proposé différentes architectures possibles correspondant à des compromis différents entre disponibilité et performance. Dans ce même article, ils ont mesuré le potentiel qu'offrait la diversité en termes de gain en fiabilité. Ils ont analysé 180 bugs dans les différents COTS utilisés et ont vérifié si ces bugs affectaient également les bases de données COTS utilisées pour lesquelles le bug n'était pas rapporté. Ils ont abouti à deux conclusions. Premièrement, la plupart des bugs ne produisent pas des défaillances franches et ne peuvent pas être tolérées sans diversité. Deuxièmement, la plupart des bugs seraient tolérés par l'utilisation de la diversité car une simple architecture comprenant deux variantes détecterait de 94% à 100% des bugs.

Même si l'utilisation de COTS en diversification fonctionnelle introduit certains problèmes, les résultats obtenus par le projet DOTS en termes de tolérance aux fautes nous conduisent à penser qu'il est possible d'utiliser des COTS en diversification fonctionnelle pour faire de la détection et de la tolérance aux intrusions.

1.2.6 La détection d'intrusions du point vue de la sûreté de fonctionnement

Nous avons vu précédemment que la sécurité fait partie intégrante de la sûreté de fonctionnement. Le projet MAFTIA [AAC+03, VNC03, WWRS03] est un projet européen qui a proposé une approche complète pour tolérer à la fois les fautes accidentelles et les fautes malignes dans des systèmes distribués à grande échelle. Le projet propose, entre autres choses, un modèle de fautes adapté au cadre de la sécurité.

Dans le cadre de la sécurité, on s'intéresse à trois attributs spécifiques de la sûreté de fonctionnement : la confidentialité, l'intégrité, et la disponibilité. Ces attributs contribuent à la définition des propriétés du système en terme de sécurité. Définir ces propriétés revient à définir la politique de sécurité du système.

Les entraves à la sécurité communément admises sont : les intrusions, les attaques, les vulnérabilités. De même que dans le domaine de la sûreté de fonctionnement, ces entraves ont un lien de causalité : une attaque exploite une ou plusieurs vulnérabilités afin de réaliser une intrusion. Ces entraves peuvent

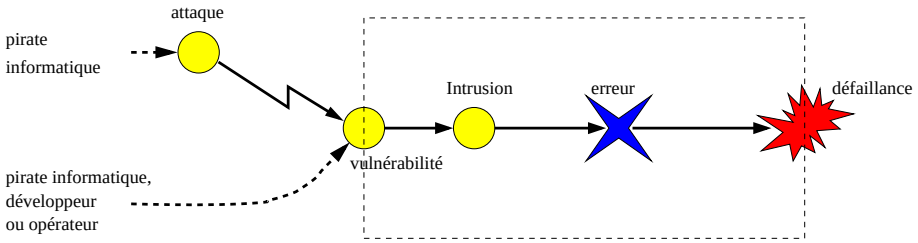


FIGURE 1.4 – Vue d'une intrusion comme une faute composée

être interprétées dans le cadre général de la sûreté de fonctionnement, et des définitions précises ont été proposées dans le cadre de MAFTIA :

- *une attaque* est une faute d'interaction externe maligne dont le but, à terme, est de violer une ou plusieurs des propriétés de sécurité (réaliser une intrusion) ;
- *une vulnérabilité* est une faute accidentelle ou intentionnelle, maligne ou non, dans les exigences, la spécification, la conception ou la configuration du système, ou dans la manière de l'utiliser, (ce qui correspond en fait à une faute de conception) qui peut être exploitée dans le but de provoquer une intrusion ;
- *une intrusion* est une faute maligne interne conséquence d'une attaque réalisée avec succès (c'est-à-dire qui a mené à une violation des propriétés de sécurité) par l'exploitation d'une vulnérabilité.

Les vulnérabilités sont les fautes essentielles présentes dans les composants, en particulier dans la conception et la configuration : fautes de développement permettant un *stack overflow*, mots de passe naïfs, ports TCP/IP non protégés, etc. Il faut remarquer qu'un attaquant peut éventuellement introduire une vulnérabilité (par exemple sous forme de logique maligne) dans le but de poursuivre son attaque globale par la suite.

Les attaques peuvent être vues de deux façons différentes : du point de vue de l'activité humaine de l'attaquant ou au niveau de l'activité observable dans le système considéré :

- attaque (point de vue humain) : une faute maligne d'interaction par laquelle un attaquant essaie de violer de façon délibérée une ou plusieurs propriétés de sécurité.
- attaque (point de vue technique) : une faute technique maligne d'interaction qui vise à exploiter une vulnérabilité dans le but d'atteindre le but de l'attaquant.

Une attaque qui a activé avec succès une vulnérabilité provoque une intrusion. Cela peut conduire à une défaillance qui est caractérisée par une erreur dans le système, comme par exemple : un compte non-autorisé avec un accès ssh, un fichier système auquel l'attaquant peut accéder à cause de la modification de droits d'accès. De tels états peuvent être corrigés ou masqués mais si rien n'est fait pour traiter les erreurs résultant de l'intrusion, cela conduit à une violation

d'une propriété de sécurité.

De même que dans le cadre général de la sûreté de fonctionnement, des moyens peuvent être mis en œuvre de manière à assurer l'obtention et le maintien des propriétés de sécurité :

- prévention des vulnérabilités et intrusions : comment empêcher l'occurrence ou l'introduction de vulnérabilités et d'intrusions ;
- tolérance aux vulnérabilités et intrusions : comment fournir un service à même de remplir la fonction du système en présence de vulnérabilités et d'intrusions ;
- élimination des vulnérabilités : comment réduire la présence de vulnérabilités ;
- prévision des vulnérabilités : comment estimer la présence, la création et les conséquences des vulnérabilités.

La détection d'intrusions entre dans le cadre de la tolérance aux intrusions au même titre que la détection d'erreurs entre dans le cadre de la tolérance aux fautes mais la détection d'intrusions couvre un plus large spectre de techniques que la seule détection d'erreurs. En effet, la détection d'intrusions peut être vue de trois manières différentes suivant qu'elle est réalisée hors-ligne, presque en temps réel ou en temps réel :

- diagnostic de faute hors-ligne (en tant que partie de la maintenance) ;
- détection d'erreur et diagnostic de fautes en ligne (de manière à assister l'administrateur de sécurité dans le traitement de faute) ;
- détection d'erreur (en tant qu'étape préalable à un recouvrement automatique d'erreur) ou détection d'erreurs et diagnostic de faute en ligne (en tant qu'étape préalable au traitement automatique de faute).

Dans le cadre du projet MAFTIA, la détection d'intrusions est définie comme l'ensemble des pratiques et techniques utilisées dans le but de détecter des erreurs pouvant conduire à des violations de la politique de sécurité, et/ou de diagnostiquer des attaques.

Dans le cadre de cette thèse, nous nous intéressons essentiellement à la première partie de cette définition car la technique de diversification fonctionnelle permet de masquer des fautes et de détecter des erreurs. Nous examinons dans la section suivante les travaux concernant la détection et la tolérance aux intrusions par diversification.

1.3 Détection d'intrusions et tolérance aux intrusions par diversification

La diversification (ici dans le sens création et utilisation d'une certaine diversité) est une technique qui a fait peu à peu sa place dans le domaine de la sécurité dans le but de limiter l'impact des vulnérabilités. Forrest et al. [FSA97] proposent d'introduire de la diversité dans les systèmes d'exploitation pour li-

miter la propagation de vers. L'homogénéité importante de l'ensemble des ordinateurs (on pourra, en effet, penser au duo : processeur Intel et systèmes d'exploitation Microsoft Windows et l'ensemble d'applications venant de Microsoft) a pour conséquence que si une erreur est présente dans un de ces systèmes, elle l'est également dans la grande majorité d'entre eux. Si une méthode d'exploitation automatique d'une vulnérabilité vient à être développée, elle pourra ainsi affecter toutes les machines ayant la même configuration. Depuis cet article, de nombreuses méthodes ont été proposées pour introduire de la diversité dans les différents systèmes à des fins de sécurité : des méthodes réarrangeant ou modifiant la mémoire [FSA97, BDS03, XKI03, BS08], des méthodes pour diversifier les appels système [CS02], des méthodes pour diversifier le jeu d'instructions [KKP03, BAF+03].

Il est alors assez naturel d'avoir vu apparaître, au début des années 2000, en partie sous l'influence de la DARPA (*Defence Advanced Research Projects Agency*) et de son programme OASIS [1] (*Organically Assured and Survivable Information Systems*) des projets proposant une architecture de type N-versions pour détecter et tolérer les intrusions. Cependant la première proposition [JA88] d'utiliser cette architecture est beaucoup plus ancienne et date de 1988. Joseph et Avizienis ont réalisé une analyse de la diversification fonctionnelle pour voir si cette technique était capable de masquer certaines intrusions en faisant le parallèle entre les fautes qui peuvent affecter la sûreté de fonctionnement du système et les intrusions qui peuvent affecter la sécurité du système.

Deux autres articles [DKL99, LS04] mettent en avant le rôle de la diversité et de la redondance dans le domaine de la sécurité.

Dans cette section, nous présentons les principes de la détection d'intrusions par diversification fonctionnelle et discutons les propriétés de tolérance aux intrusions qu'une telle approche procure. Nous présentons les différents travaux proches des nôtres et en quoi notre travail se différencie de ces travaux.

1.3.1 Principes de la détection et de la tolérance

L'idée principale repose sur l'exécution en parallèle d'un service, souvent sur des machines différentes, par des COTS souvent distincts. Le principe de la détection et de la tolérance repose sur le même principe que celui de la programmation N-versions. L'hypothèse essentielle pour obtenir les propriétés de détection et de tolérance aux intrusions est celle de l'indépendance des vulnérabilités, c'est-à-dire que les différents serveurs n'ont pas les mêmes vulnérabilités. Sans aucune diversification, il n'est pas possible de détecter ou de tolérer les intrusions car toutes les vulnérabilités sont corrélées et entraînent des erreurs similaires.

Il faut ici revenir sur le concept de vulnérabilité. Il est, dans notre cas, contraire au sens commun. Au sens commun du terme, une vulnérabilité affecte un logiciel dans une ou plusieurs versions, une ou plusieurs configurations, un ou plusieurs contextes d'exécutions (processeur, système d'exploitation par

1. `http://www.tolerantsystems.org/oasis.html`

exemple), etc. Par exemple (fictif), une vulnérabilité peut être un dépassement de tampon dans les versions d'Apache inférieures à 1.2.10 quelque soit la configuration et le contexte d'exécution. Dans notre cas, nous différencions les vulnérabilités suivant la version du logiciel, sa configuration et son contexte d'exécution. Nous allons expliciter cette considération sur un exemple. Il est possible de faire tourner un serveur HTTP comme Apache sur des systèmes d'exploitation différents (Linux et MacOS-X) et des architectures de processeurs différents (x86 et PowerPC). Si le code source d'Apache contenait une vulnérabilité (ici au sens commun du terme) de type dépassement de tampon, la vulnérabilité serait présente dans les deux serveurs. Cependant, il serait hautement improbable de construire une même requête HTTP exploitant cette vulnérabilité sur les deux serveurs à la fois car le code injecté pour exploiter la vulnérabilité dépend à la fois du jeu d'instructions du processeur et de l'architecture du système d'exploitation (par exemple, la manière dont sont appelés les appels système, la convention pour le passage des arguments des appels système, ...). Dans ce cas, nous considérons que les serveurs sont bien affectées par deux vulnérabilités différentes qui vont conduire à des erreurs distinctes en cas de tentative d'injection de code.

Il est possible de conserver des capacités de détection même si l'attaquant est capable d'exploiter des vulnérabilités dans les différentes versions grâce à une même entrée. C'est le cas si l'intrusion entraîne un comportement différent des différentes versions. Il n'est éventuellement pas possible de tolérer dans ce cas : par exemple, si les réponses des variantes sont toutes différentes.

Nous reviendrons de manière plus formelle et plus en détails sur ces concepts dans 2.1.2.2.

Les différents projets qui utilisent la diversification pour réaliser de la détection d'intrusions peuvent être classés en deux grande catégorie : l'approche de type boîte noire et l'approche de type boîte grise. L'approche boîte noire consiste à considérer les COTS comme des boîtes noires et à n'observer que les événements se déroulant au niveau de l'interface standardisée. Les projets qui utilisent une approche de type boîte grise inspectent le comportement des COTS à un niveau plus bas : au niveau des appels système ou des accès mémoire, par exemple.

1.3.2 Les approches de type boîte noire

Plusieurs approches de type boîte noire ont été proposées, la moitié étant proposée dans le cadre du programme OASIS de la DARPA. Toutes ces approches proposent d'utiliser différents COTS pour protéger le service, ce n'est pas le cas de certaines approches de type boîte grise. Les approches de type boîte noire sont sensibles aux intrusions qui n'affectent pas les sorties des différents COTS ou celles où l'attaquant est capable de prendre le contrôle du flux d'exécution du programme et ainsi éventuellement de faire en sorte que la sortie du COTS compromis soit identique (du point de vue de l'algorithme de comparaison) à celles des autres COTS. Pour compenser ce point faible, certains projets

introduisent d'autres IDS au niveau hôte pour détecter les intrusions que leur approche de type boîte noire pourrait manquer.

1.3.2.1 Le projet BASE

Le projet BASE (*BFT with Abstract Specification Encapsulation*, BFT étant une bibliothèque développée précédemment par les auteurs et qui signifie *Bizantine Fault Tolerance*) [RCL01, CRL03] est un projet qui propose d'utiliser un modèle abstrait du comportement du service protégé dans le but de permettre l'utilisation de COTS et ainsi de réduire le coût de la tolérance aux fautes byzantines et d'améliorer la capacité à masquer les erreurs. Les auteurs ont développé un prototype de service NFS (*Network File System*) et pour une base de données objet et évoquent la possibilité d'utiliser leur approche pour des bases de données relationnelles. Le modèle abstrait décrit le fonctionnement normal du service diversifié et masque les différences de spécification et de conception entre les COTS, comme, par exemple, le descripteur de fichier dans le système NFS proposé grâce à des wrappers de conformité (*conformance wrappers*). Ces wrappers sont utilisés pour normaliser les sorties des différentes implémentations du service également dans le cas du non-déterminisme liée à l'approche, comme, par exemple, pour les timestamps de dernière modification d'un fichier.

Cette approche peut être utilisée dans la cas où l'interface du service est bien documentée et bien définie. C'est le cas pour NFS mais cela est moins vrai pour les serveurs web : il est à notre avis très difficile de construire un modèle abstrait complet pour les serveurs web ; de plus, ce modèle va dépendre de l'application web fonctionnant sur le serveur. Il en est de même pour les wrappers de conformité.

Les auteurs étudient également de manière approfondie et théorique une approche proactive de rajeunissement logiciel grâce au modèle abstrait défini et des points de reprise.

Les auteurs sont les premiers à avoir clairement mis en avant les problèmes liés au non-déterminisme et aux différences de spécification entre les différentes implémentations utilisées.

Les tests menés dans le cadre de ce projet ne s'intéressent qu'aux performances de l'architecture mise en place et ne mesurent pas la capacité de détection ni le nombre des faux positifs et le coût éventuel de ces faux positifs en terme de recouvrement.

1.3.2.2 Le projet HACQIT

Le projet HACQIT (*Hierarchical Adaptive Control for QoS Intrusion Tolerance*) a été présenté dans les articles [RJL+02, JRC+02, RJCM03]. Les auteurs proposent un système tolérant aux intrusions fondé sur la détection de nouvelles intrusions en généralisant les signatures de certaines attaques. Le système est capable de détecter de nouvelles attaques grâce à l'utilisation d'une paire de serveurs web COTS (un serveur IIS et un serveur Apache) en comparant les réponses données par ces deux serveurs. Une différence entre les deux réponses

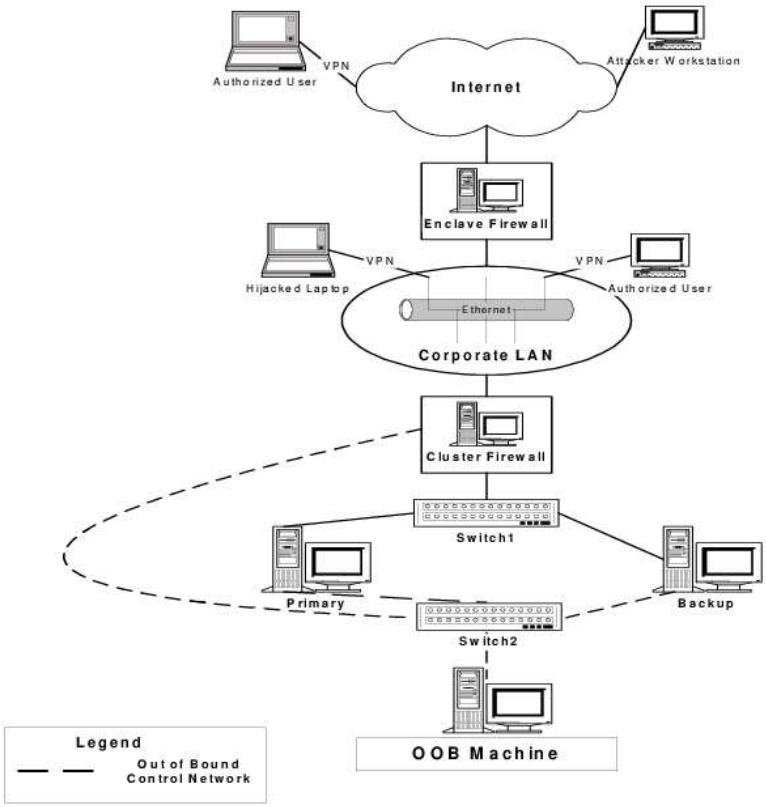


FIGURE 1.5 – Schéma général de l'architecture de HACQIT

signifie la présence d'une attaque. Sur le serveur primaire comme sur le serveur de backup, sont installés un logiciel de détection d'intrusions réseaux (en l'occurrence Snort [Roe99]), un programme moniteur d'hôte, un programme moniteur applicatif, un logiciel de contrôle d'intégrité (Tripwire [Tri]) ; une alerte peut également être levée par un de ces outils et transmise au « Out-of-Band Controller » qui est chargé de gérer les différentes alertes. Il peut en effet changer les règles du pare-feu, échanger le rôle du primaire et du backup, généraliser la signature d'une nouvelle attaque détectée grâce à l'utilisation d'une sandbox. Des techniques de rajeunissement sont également utilisées lorsqu'une erreur a été détectée sur un serveur pour le remettre dans sa configuration d'origine.

La figure 1.5 montre l'architecture générale du projet HACQIT. La requête d'un client est transmise au serveur primaire. Celui-ci la relaie au « Out-of-Band

Controller » qui est chargé de décider si la requête est correcte ou non. Ce choix est fondé sur la comparaison de la requête avec une liste de requêtes étiquetées comme des intrusions. Si la requête correspond à une requête présente dans la liste, elle est rejetée sinon, elle est fournie au serveur IIS et au serveur Apache. Si les réponses sont distinctes, le serveur considéré comme fautif est relancé et le système essaie d'établir une signature de l'attaque à partir des dernières requêtes.

La détection des différences entre les réponses des serveurs est assez restreinte ; elle prend en compte la présence ou non de réponse (dépassement de délai d'un des deux serveurs). Si les deux serveurs ont répondu, elle se focalise sur le code HTTP retourné et deux cas sont seulement pris en compte (combinaison 2XX/4XX et combinaison 2XX/3XX). De plus, l'utilisation de deux serveurs uniquement ne permet pas réellement de détecter lequel des deux serveurs a été attaqué.

Le système est capable de développer de nouvelles signatures d'attaques automatiquement. En effet, lorsqu'une intrusion est détectée par l'un des IDS ou grâce à la diversification, toutes les requêtes depuis le dernier rajeunissement sont rejouées dans une sandbox pour déterminer quelles requêtes sont responsables de l'intrusion. Différentes étapes sont suivies pour établir une règle de filtrage de ce type de requêtes plus générales que les requêtes incriminées.

Les auteurs n'ont pas évalué en pratique l'apport de la diversification en termes de détection d'intrusions et ont cherché à évaluer le système de test et de généralisation proposé pour bloquer une nouvelle attaque détectée.

1.3.2.3 Le projet SITAR

Le projet SITAR [WGS+01, WWB01, WMT03] (*Scalable Intrusion Tolerant ARchitecture*) est un autre projet proposant une architecture tolérante aux intrusions. Ils mettent également en œuvre des techniques de redondance, de diversité et de rajeunissement pour garantir la tolérance. L'architecture proposée est relativement proche de celle que nous proposons. La figure 1.6 montre cette architecture. Il est à noter que cette architecture n'est pas spécifique aux serveurs web.

Le service proprement dit est assuré par des serveurs COTS vulnérables à des intrusions. Les serveurs proxy sont les points d'entrée du système. Toutes les requêtes arrivent sur l'un des serveurs proxy. Celui-ci assure la politique de service suivant la stratégie de tolérance aux intrusions en cours. La politique de service détermine à quels serveurs COTS la requête doit être envoyée et quelle stratégie est mise en place pour décider quelle réponse doit être renvoyée finalement au client. Les requêtes sont traitées tout d'abord par les moniteurs d'acceptation (*acceptance monitors*). Si elles sont valides, elles sont ensuite transférées aux serveurs COTS. Les réponses des serveurs COTS sont de nouveau traitées par les moniteurs d'acceptation. Ceux-ci vérifient la validité des réponses suivant différents critères puis transfèrent les réponses avec une analyse de la validité des réponses aux *ballot monitors*. Si une réponse ne vérifie pas les critères de

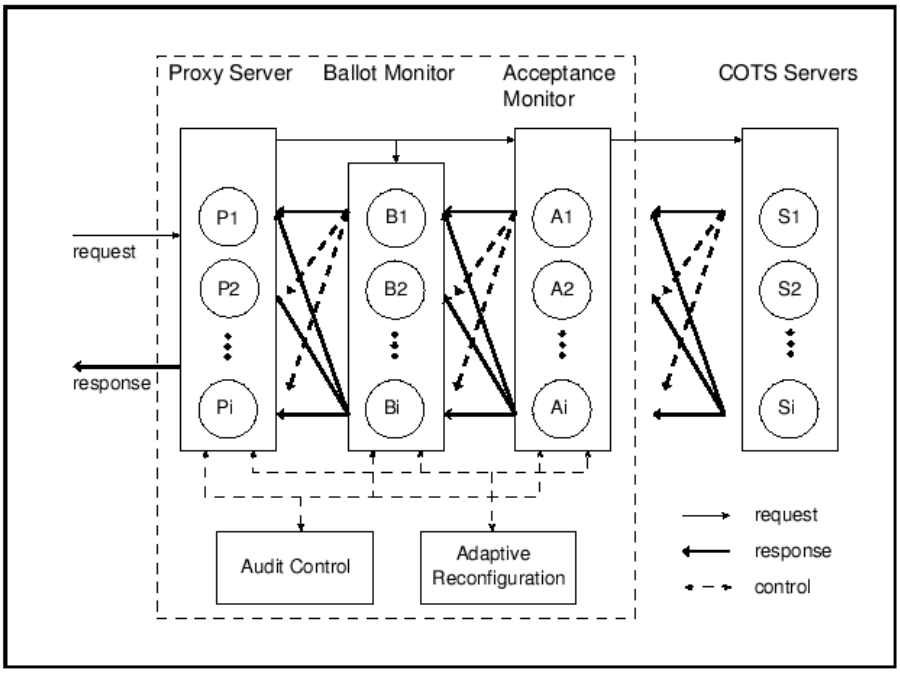


FIGURE 1.6 – Schéma général de l'architecture de SITAR

validité testés, ils envoient également une alerte au module de reconfiguration dynamique (*adaptive reconfiguration module*) signalant la présence d'une intrusion. Les *ballot monitors* reçoivent les réponses des moniteurs d'acceptation. Différentes configurations sont proposées : soit un *ballot monitor* est associé à un serveur COTS et reçoit donc la réponse d'un seul moniteur d'acceptation, soit un ballot monitor reçoit les réponses de tous les moniteurs d'acceptation. Dans le premier cas, il est nécessaire d'effectuer un protocole d'accord entre les différents *ballot monitors*. Dans le second cas, chaque *ballot monitor* est capable de décider quelle réponse envoyer au serveur proxy. Cette réponse peut être envoyée par un seul *ballot monitor* désigné lors du lancement du système, par un *ballot monitor* désigné aléatoirement au cours de l'exécution ou encore tous les *ballot monitors* envoient leur version du résultat et laissent le serveur proxy décider de la réponse finale.

Deux autres modules sont présents dans l'architecture : le module de contrôle (*audit control module*) et le module de reconfiguration dynamique (*adaptive reconfiguration module*). Le module de contrôle vérifie les fichiers d'audit générés par chacun des différents composants de l'architecture : serveur proxy, ballot monitor, moniteur d'acceptation et le module de reconfiguration dynamique.

Il exécute périodiquement des suites de tests sur chacun des composants pour vérifier leur comportement. Les résultats de ces tests et de l'analyse des fichiers d'audit sont envoyés au module de reconfiguration dynamique qui va décider de la stratégie à adopter. C'est en effet ce module qui décide du nombre de serveurs COTS interrogés, de la configuration des ballot monitors. Ce module sert à adapter le niveau de performances au degré de danger actuel. En effet, si aucune attaque n'a été détectée depuis un certain temps, il n'est pas nécessaire de maintenir un haut niveau de redondance pour les serveurs COTS, par exemple, car cela entraîne une dégradation des performances du système qui n'est pas nécessaire dans la configuration actuelle.

La détection des différences entre les réponses des différents serveurs se fait par transformation des données suivant trois méthodes en fonction du niveau de sécurité désiré :

- Fletcher checksum [Mck84] ;
- MD5 checksum [Riv92] ;
- keyed MD5 checksum.

Cela permet des comparaisons rapides des résultats notamment lorsque la taille des données est importante mais manque certainement de finesse et ne permet pas de réaliser un diagnostic des différences. Ce projet manque également pour l'instant de résultats expérimentaux qui valideraient l'approche et l'architecture.

1.3.2.4 Le projet DIT

DIT (*Dependable Intrusion Tolerance*) [VAC+02, SDN03] est un projet proposant également une architecture pour un système tolérant aux intrusions. Cette architecture est très similaire à celle proposée par SITAR. La figure 1.7 présente l'architecture générale proposée.

L'architecture se compose d'un ensemble de serveurs proxy redondants qui transfèrent les requêtes à un ensemble de serveurs applicatifs redondants. Ici aussi, l'architecture proposée n'est pas spécifique aux serveurs web. La configuration entière est surveillée par divers mécanismes assurant l'intégrité des serveurs dont un système de détection d'intrusions basé sur le *framework* EMERALD [PN97]. Les serveurs d'application ainsi que les serveurs proxy sont diversifiés : ils fournissent le même service mais ont été développés de manière indépendante. Ainsi, la probabilité qu'ils soient vulnérables à la même attaque est moins importante. Les serveurs d'application peuvent utiliser des COTS ; cela permet de réduire le coût de l'architecture. 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 décider d'utiliser des protocoles d'accord plus stricts, de filtrer les requêtes provenant de clients suspects ou encore de redémarrer des serveurs ou des services qui sont compromis.

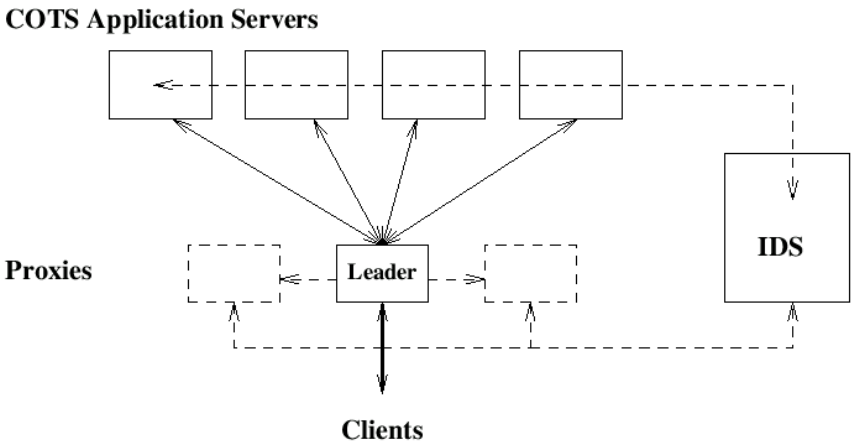


FIGURE 1.7 – Schéma général de l'architecture de DIT

Les serveurs d'application sont pourvus de moniteurs d'hôtes et exécutent principalement des COTS. Les serveurs proxy sont l'interface avec les serveurs d'application. Ils reçoivent les requêtes des clients et les transfèrent aux serveurs d'application, surveillent l'état de ces derniers et également des autres serveurs proxy et adaptent dynamiquement la politique du système en fonction des alertes émises par le sous-système de surveillance. Un proxy est désigné comme leader ; il est chargé de filtrer et de transférer les requêtes à un ou plusieurs serveurs d'application, de mettre en œuvre un mécanisme de décision, dépendant du régime actuel, pour le choix de la réponse à envoyer au client, et également de répartir la charge entre les différents serveurs d'application. Les autres serveurs proxy sont optionnels et sont utilisés pour surveiller toutes les communications entre le serveur leader et les serveurs d'application. Ils sont eux-mêmes surveillés par les autres serveurs proxy et le sous-système de surveillance.

L'architecture fonctionne de la manière suivante :

1. Le serveur proxy leader accepte une requête, la vérifie et la rejette dans le cas où celle-ci est malformée.
2. Il la transfère à un certain nombre de serveurs d'application en fonction de la politique actuelle.
3. Les serveurs d'application traitent la requête et retournent les résultats au serveur proxy leader. Si un accord suffisant est trouvé entre les différentes réponses, le serveur renvoie le contenu au client.
4. La politique du système est ajustée en fonction de la présence ou non d'alertes émises par le sous-système de surveillance et suivant les réponses

envoyées par les différents serveurs d'application.

5. Les serveurs proxy auxiliaires, s'il y en a, vérifient la transaction pour assurer que le comportement du leader est correct.

Pour assurer la tolérance aux intrusions, d'autres techniques sont proposées dans l'architecture. Un protocole de challenge-réponse effectué périodiquement qui permet de vérifier le fonctionnement à la fois des serveurs d'application et des serveurs proxy. Ce protocole peut être engagé par un serveur proxy quelconque. La vérification en ligne des serveurs proxy permet d'assurer que ceux-ci se comportent conformément à leur spécification. Une technique de rajeunissement (redémarrage des machines et/ou des services) des serveurs proxy et des serveurs d'application permet d'empêcher la baisse de fiabilité introduite par l'utilisation prolongée du service et d'empêcher des intrusions incrémentales longues à se mettre en place.

Une implémentation a été menée dans le cadre de serveurs web. La comparaison des réponses se fait grâce au calcul d'une somme MD5. Un seul serveur d'application renvoie la somme MD5 et le contenu, les autres se contentant d'envoyer la somme MD5. Cela permet d'économiser de la bande passante et permet une comparaison rapide dans le cas d'un contenu important. Le serveur proxy leader calcule la somme MD5 du contenu reçu et la compare à celle envoyée avec celui-ci et aux autres sommes MD5. Ce type de comparaison ne permet pas par contre de faire du diagnostic. De plus, les expérimentations menées sont assez succinctes et ne mettent pas en exergue le fait que la comparaison des réponses des différentes versions permet de détecter de nouvelles attaques. En effet, les sous-systèmes de détection d'intrusions utilisés comportent la signature de l'attaque utilisée comme exemple (le ver informatique Code Red).

1.3.2.5 Discussion

L'architecture de l'ensemble de ces projets est sensiblement la même que celle que nous utilisons : elle est fondée sur l'architecture classique utilisée en programmation N-versions. Cependant, plusieurs points différencient notre approche de celles présentées ci-dessus.

Ces 4 projets sont essentiellement des projets orientés vers la tolérance aux intrusions alors que notre objectif est d'avantage d'évaluer les capacités de détection d'intrusions d'une approche fondée sur la diversification fonctionnelle.

À l'exception du projet BASE, les autres projets mettent en place d'autres systèmes de détection d'intrusions dans le but de détecter au plus tôt les éventuelles intrusions. Pour détecter les intrusions qui ne seraient pas visibles au niveau des sorties des COTS, des IDS au niveau hôte sont mis en place sur chaque serveur COTS. Les projets HACQIT et DIT ajoutent également un IDS au niveau réseau (Snort [Roe99]). À l'instar de BASE, nous proposons seulement une architecture de détection d'intrusions fondée sur la diversification fonctionnelle.

Du fait de l'orientation tolérance aux intrusions de ces 4 projets, aucune évaluation en termes de faux positifs et de faux négatifs n'a été faite. Des études

théoriques ont été faites pour évaluer les capacités de détection, c'est-à-dire évaluer les intrusions qui peuvent être détectées et tolérées. Aucune évaluation quantitative des faux positifs n'a été faite par contre. Ceux-ci ont un impact potentiel sur l'architecture : choix du niveau de sécurité dans SITAR et DIT, recherche de nouvelles attaques et filtrage potentiel de requêtes normales pour HACQIT, reconfigurations non nécessaires des serveurs. Dans cette thèse, nous avons cherché à évaluer les capacités de détection de cette architecture et tester notre approche en termes de faux positifs de l'approche.

1.3.3 Les approches de type boîte grise

Les approches de type boîte grise, contrairement aux approches de type boîte noire, sont potentiellement capables de détecter les intrusions qui n'influencent pas les sorties normalisées des COTS, notamment celles par lesquelles l'attaquant est capable de prendre le contrôle du flux d'exécution de l'application. Elles sont moins naturelles que les approches de type boîte noire où le choix des éléments comparés est dicté par l'interface standardisée commune à tous les COTS. Dans les approches de type boîte grise, les éléments comparés sont internes aux différents COTS et il n'y aucune raison pour que ceux-ci soient directement comparables. Deux types d'approches ont été proposées pour résoudre le problème de la comparaison :

- Certains travaux [BZ06, CEF+06, NTEK+08, SGJ+08, SGF08] ont proposés d'utiliser le même COTS mais diversifié grâce à des techniques de diversification automatique, essentiellement de la position en mémoire des données. Cela permet de résoudre les problèmes de comparaison puisque que le code exécuté est le même. Cela permet également de compenser certains problèmes liés au non-déterminisme. Le désavantage éventuel est une moins grande couverture de détection. Ces approches se concentrant souvent sur un type d'exploitation de vulnérabilités.
- Gao, Song et Reiter [GRS05, GRS06a] proposent de leur côté un mécanisme fondé sur l'apprentissage pour effectuer la comparaison des séquences d'appels système entre des COTS différents.

Nous avons également proposé une approche de type boîte grise au cours de cette thèse : elle est plus proche des travaux menés par Gao au cours de sa thèse dans le sens où nous avons choisi d'utiliser des COTS différents et non de diversifier automatiquement les COTS. Elle se différencie par l'utilisation d'une abstraction, les graphes de flux d'informations, pour comparer le comportement des COTS plutôt que de reposer sur un apprentissage.

Dans la suite, nous détaillons les différents projets utilisant une approche de type boîte grise.

1.3.3.1 DieHard

Berger et Zorn [BZ06, BZ07] ont proposé DieHard qui est un *framework* d'exécution redondante où chaque réplique utilise un agencement différent et

aléatoire des objets à l'intérieur du tas, tout ceci dans le but d'offrir une protection probabiliste contre les erreurs affectant la mémoire. Leur méthode repose sur un nouvel allocateur de mémoire, nommé DieFast, plaçant les objets sur le tas de manière à les éloigner le plus possible et ainsi éviter que certaines erreurs de programmation n'affectent un autre objet. Leur allocateur de mémoire permet, avec une probabilité élevée, de masquer cinq types d'erreurs de mémoire sur les six possibles sur le tas sans utiliser plusieurs variantes. Seules les lectures sur du contenu non initialisé requièrent la présence de plusieurs variantes pour être détectées.

DieHard passe toutes les entrées aux différentes variantes et récupère les sorties des variantes. La comparaison se fait sur les sorties des variantes quand les buffers de sortie sont remplis. Si ces buffers contiennent les mêmes données, DieHard écrit un des buffers de sortie. Sinon, il y a une forte probabilité qu'une seule des variantes ait un buffer de sortie différent, cette variante est tuée par DieHard et le nombre de variantes est diminué de un.

Les auteurs effectuent une analyse théorique des probabilités de détection et de masquage de certaines erreurs. Ils effectuent également des tests en injectant des fautes dans un programme et montrent aussi que leur allocateur est capable de détecter et tolérer une attaque exploitant un *buffer overflow* dans le proxy Squid.

Novark, Berger et Zorn [NBZ07] ont proposé Exterminator qui repose également sur DieFast. Exterminator reprend les mêmes idées que DieHard mais permet le masquage des fautes de programmation en proposant des correctifs binaires. Ces correctifs sont mis en place par un allocateur correctif qui permet de prendre en compte les dépassements de tampons notamment en allouant un espace plus grand pour l'objet affecté.

1.3.3.2 Le projet N-variant

Le projet N-variant [CEF+06, NTEK+08] utilise une approche similaire à celle de DieHard. Au lieu d'utiliser différents COTS pour assurer l'indépendance des fautes, les auteurs utilisent des techniques automatiques de diversification (partitionnement de l'espace d'adressage, étiquetage des instructions par exemple) pour construire à partir d'un seul COTS des variantes diversifiées de ce COTS. Ces variantes présentent les mêmes vulnérabilités mais ces vulnérabilités ne peuvent pas être exploitées par exactement la même attaque. Chaque variation permet de détecter de manière certaine un certain type d'attaques : si l'espace d'adressage global est partitionné et que chaque variante occupe une partition de cet espace d'adressage, toutes les attaques nécessitant une adresse absolue vont échouer de manière certaine dans au moins une des variantes.

Comme dans le cas de DieHard, les variantes sont exécutées sur la même plate-forme. Le prototype développé modifie le noyau du système d'exploitation pour s'assurer que les différentes variantes exécutent les mêmes appels système dans le même ordre. Même si cette approche résout certains problèmes liés au non-déterminisme, il en reste certains liés par exemple à l'utilisation des signaux POSIX. Certains appels système sont considérés comme dangereux car ils

peuvent impliquer des canaux cachés (*mmap*) ou de sortir du schéma redondant (*execve*).

Cette approche présente cependant un inconvénient : il est nécessaire de construire des variantes pour chaque type d'attaques que l'on souhaite détecter. Sans hypothèse supplémentaire, pour être sûr de détecter une classe d'attaques, il est nécessaire d'avoir deux variantes qui ne diffèrent que par la variation permettant de détecter cette classe d'attaques : pour deux classes d'attaques, il faudra donc trois variantes ; de manière générale, pour n variations binaires, il faut $n + 1$ variantes.

Ce projet se différencie de DieHard par le fait que la protection apportée n'est plus probabiliste mais certaine ; elle ne s'applique par contre qu'à certaines classes d'attaques.

1.3.3.3 Travaux de Babak Salamat

Les travaux de Babak Salamat [SGJ+08, SGF08] portent sur l'exécution de multiples variantes d'un même programme dans un environnement multi-cœur. Les variantes se différencient par le positionnement des différents éléments de la mémoire : la pile est renversée dans l'une des deux variantes. Cela assure notamment qu'en cas de dépassement de tampon, le comportement des serveurs sera différent rendant possible la détection. La synchronisation et la comparaison se font au niveau des appels système : les variantes doivent exécuter les mêmes appels système avec des arguments équivalents.

Ce projet est donc très proche du projet N-variant : il se heurte aux mêmes difficultés techniques : gestion des signaux, de l'appel système *mmap*. Les différences se situent essentiellement d'une part, dans la variation appliquée et d'autre part, dans les techniques utilisées : le moniteur est, dans le cas de ces travaux, un processus utilisateur classique et non lié au noyau comme dans le cas du projet N-variant et les variations sur la pile sont également réalisées par le compilateur alors que les variantes sont créées soit grâce à un script gérant la liaison des objets soit par de la réécriture de binaire, pour le projet N-variant.

1.3.3.4 Distance comportementale

Les travaux menés lors de la thèse de Gao [Gao07] sont les plus proches des travaux que nous avons menés sur notre seconde approche de détection d'intrusions par diversification fonctionnelle. Contrairement aux autres approches de type boîte grise présentées ci-dessus, Gao, Reiter et Song [GRS05, GRS06a, GRS06b] proposent l'utilisation de différents COTS dans un système de détection d'intrusions par diversification fonctionnelle de type boîte grise. Les auteurs justifient leurs travaux par la possibilité de détecter les *mimicry attacks* [TKM02, WS02] qui affectent les approches classiques surveillant les appels système de processus.

La difficulté dans cette approche est le choix des éléments de comparaison et la méthode de comparaison. Il n'est pas aisé de comparer le fonctionnement interne de deux applications différentes. Gao, Reiter et Song ont choisi de com-

parer les séquences d'appels système effectuées par les différents COTS. Ces appels système ne sont pas directement comparables car les COTS sont des applications différentes avec une conception interne différente. Même s'ils fonctionnent sur le même système d'exploitation, les COTS ne vont pas exécuter les mêmes appels système avec les mêmes arguments. Si les systèmes d'exploitation sur lesquels fonctionnent les différents COTS sont eux aussi diversifiés, la comparaison est encore plus ardue. En effet, il n'y a pas de correspondance directe entre les appels système de plusieurs systèmes d'exploitation : un appel système sur un système d'exploitation S_1 peut correspondre à un ou plusieurs appels système sur un système d'exploitation S_2, voire aucun ou encore un ou plusieurs appels système mais avec des paramètres particuliers.

Les auteurs de ces travaux proposent une manière de comparer des suites d'appels système effectués par des COTS sur différents systèmes d'exploitation. Ils introduisent la notion de distance comportementale qui est une mesure de la déviation des comportements de deux processus. Ils proposent deux manières d'évaluer cette distance : en utilisant une distance évolutionnaire [GRS05] ou des modèles de Markov cachés [GRS06a]. Une intrusion devrait modifier le comportement de l'un des deux processus et devrait donc accroître la distance comportementale. Si la distance calculée est au-dessus d'un certain seuil, une alerte est émise.

L'architecture présentée par les auteurs consiste en l'exécution de COTS différents ou identiques sur des systèmes d'exploitation différents. Les auteurs ont implémenté deux prototypes de leur architecture : un prototype pour serveurs web avec différents serveurs web (Apache, Abyss et MyServer) fonctionnant sous Linux et sous Windows et un prototype d'un serveur de jeu en ligne écrit en Java fonctionnant à la fois sous Linux et Windows. En ce qui concerne le serveur de jeu, c'est le même code qui est exécuté sur les deux systèmes d'exploitation. L'utilisation de deux systèmes d'exploitation différents a de bonnes chances de détecter les attaques ayant pour objectif l'injection de code binaire. Les attaques exploitant des fautes de logique dans le serveur ne pourront être détectées que si elles sont spécifiques à un système d'exploitation. Les auteurs ont également fait des tests plus poussés en utilisant seulement un serveur web Apache [GRS06b]. Les versions Windows et Linux d'Apache sont conçues à partir de codes sources différents en partie au moins. Les auteurs affirment qu'ils peuvent être ainsi considérés comme des COTS différents.

Les auteurs ont effectué des tests en terme de faux positifs pour leurs deux prototypes. Ils ont fixé la distance au-delà de laquelle les séquences d'appels système sont considérées comme différentes, ce qui serait symptomatique d'une attaque, grâce à l'évaluation de la distance qu'aurait la meilleure attaque *mimicry* possible. Cette évaluation est théorique et assez limitée : la meilleure attaque *mimicry* doit contenir un appel système *open* et un appel système *write*. Ils obtiennent alors des résultats relativement bons en termes de faux positifs : dans le cas de la distance évolutionnaire, dans la pire configuration (couple de serveurs web différents), il y a 2,08% de faux positifs et dans le cas des modèles de Markov cachés, dans la pire configuration (couple de serveurs web différents), il y a 1,44% de faux positifs. Pour le serveur de jeu, 14 faux positifs ont été

comptabilisés pour 39 000 événements de jeu.

Pour améliorer les performances de leur approche, les auteurs proposent d'utiliser des machines virtuelles pour exécuter les deux systèmes d'exploitation et les COTS. Ceci a notamment pour but d'éviter les latences dues au réseau. Tous les transferts entre les machines virtuelles se font en effet grâce à des copies en mémoire. Bien que nous ayons choisi dans notre implémentation d'utiliser des machines réelles, cette technique peut être utilisée également dans notre approche. L'avantage de machines réelles est double : d'une part, d'avoir de la redondance physique et donc d'être capable de tolérer des fautes physiques et d'autre part, il est possible de diversifier, sans perdre trop de performances, les processeurs ce qui rend peu probable le fait qu'une injection de code binaire fonctionne sur deux machines à la fois.

Un inconvénient de l'approche décrite dans ce paragraphe est qu'elle ne considère que le numéro des appels système, alors que l'opération effectuée par un appel système dépend souvent des paramètres qui lui sont passés. Il est ainsi possible d'utiliser ce fait pour éviter la détection.

1.3.3.5 Discussion

Deux grands types d'approches ont été proposés pour la détection d'intrusions par diversification de type boîte grise : d'une part, les travaux utilisant les mêmes COTS et des techniques de diversification automatique et, d'autre part, les travaux menés par Gao, sur la comparaison de séquences d'appels système effectués par différents COTS.

Les premiers travaux permettent une granularité beaucoup plus faible : il est possible de comparer instruction par instruction l'exécution des variantes. Cela aurait cependant un coup prohibitif en termes de performance. Il est plus raisonnable de comparer les appels système et leurs arguments comme l'ont choisi Cox et al. [CEF+06] et Babak et al. [SGJ+08] ou alors les buffers d'entrées/sorties comme dans le cas du projet DieHard [BZ06]. Il est alors possible de détecter des intrusions de manière assez précise tout en limitant fortement le nombre de faux positifs dus aux différences de spécification s'ils utilisaient des COTS différents. Un des inconvénients de ce type d'approche est le fait qu'elles sont toutes destinées à détecter un type particulier d'erreurs ou d'intrusions. DieHard ne s'attache qu'aux erreurs en mémoire sur le tas (ce qui comprend les attaques de type *heap overflow*), Babak et al. s'intéressent aux attaques en mémoire ayant une influence sur la pile du programme et Cox et al. proposent deux variations qui permettent de détecter les intrusions nécessitant l'utilisation d'une adresse absolue et les intrusions injectant du code binaire.

Le taux de couverture de l'approche suivie par Gao, Reiter et Song, qui est semblable à la nôtre, est potentiellement plus important. Il dépend entièrement de la diversification entre les COTS, au niveau applicatif, des systèmes d'exploitation et du matériel. Cette approche est cependant probabiliste dans le sens où des vulnérabilités communes peuvent exister et conduire à leur exploitation simultanée sur les différents COTS, qui pourrait ne pas être détectée. Cette approche présente également un taux de faux positifs potentiellement plus élevé :

les COTS étant différents, tout comme les systèmes d'exploitation et le matériel, des différences de spécification et de conception peuvent naturellement conduire à des comportements différents suivant les COTS. Gao, Reiter et Song ont choisi de résoudre ce problème par une phase d'apprentissage où le système apprend les correspondances entre les séquences d'appels système. Leur démarche est donc sensible à toutes les critiques concernant l'apprentissage (voir 1.1.1.2). Nous avons choisi d'utiliser une abstraction pour modéliser ce comportement et comparer le comportement des différents COTS au niveau des appels système. Notre méthode peut éventuellement s'accompagner d'une phase d'apprentissage des correspondances, qui pourrait réduire le nombre de faux positifs.

1.4 Résumé

Bien que l'approche de détection d'intrusions par signature montre de bons résultats en terme de pertinence, elle présente des défauts inhérents : elle est incapable de détecter de nouvelles attaques ou intrusions. En choisissant de modéliser le comportement normal de l'entité, il est possible de détecter de nouvelles attaques ou intrusions ; c'est ce qui est fait dans l'approche comportementale. Cette approche se heurte à certaines difficultés : choix des éléments du modèle de comportement, taux de faux positifs généralement élevé.

Parallèlement à ces approches dites classiques, un rapprochement s'est effectué entre le domaine de la sécurité et celui de la sûreté de fonctionnement : dès les années 1980, certains chercheurs [RD86, JA88] ont affirmé les bénéfices pouvant être apportées par certaines techniques de la tolérance aux fautes dans le domaine de la sécurité. Plus récemment, par l'intermédiaire du projet MAFTIA et du programme OASIS, des ponts ont été créés entre les deux domaines. Notre démarche entre pleinement dans ce cadre : nous utilisons une technique de tolérance aux fautes pour détecter les intrusions.

D'autres projets, notamment dans le cadre du programme OASIS, ont déjà exploré des approches de détection et de tolérance aux intrusions utilisant une approche reposant sur la diversité et la redondance. Ces projets se distinguent suivant l'approche choisie : de type boîte noire ou de type boîte grise.

Les projets de type boîte noire se sont essentiellement consacrés à la tolérance aux intrusions de leur architecture. Très peu d'évaluations ont été faites sur les faux positifs générés par l'architecture que ce soit de manière théorique ou pratique, excepté dans le cadre du projet BASE, où les auteurs proposent un modèle abstrait permettant de masquer les différences de spécification. Notre première méthode est une approche de type boîte noire et entre plus dans le cadre de la détection d'intrusions et nous avons cherché à évaluer l'efficacité de l'approche par diversification en termes de faux positifs notamment.

Les projets suivant une approche de type boîte grise peuvent être classés en deux parties : les projets utilisant des techniques de diversification automatique et les travaux de Gao proposant l'utilisation de diversité naturelle. Notre seconde

méthode de détection se rapproche des travaux de Gao mais ne repose pas sur l'apprentissage des correspondances entre les séquences d'appels système mais sur une abstraction sous la forme de graphes de flux d'informations. Cette abstraction permet d'apporter plus de sémantique à la détection et peut fournir une aide à un premier diagnostic, facilitant ainsi le travail de l'opérateur.

Chapitre 2

Modèles et algorithmes de détection

« Les rapports étaient tous différents, conclut Anderton. Tous uniques. Mais deux d'entre eux étaient d'accord sur un point : si je restais en liberté, je tuais Kaplan. C'est ce qui a créé l'illusion d'un rapport majoritaire. Car ce n'était que cela - une illusion. »

Philip K. Dick, Rapport minoritaire.

La détection d'intrusions comportementale repose sur la comparaison du comportement de l'entité surveillée avec un modèle du comportement correct de cette entité. Ce modèle de comportement correct est difficile à établir explicitement car il faut que le modèle soit complet, c'est-à-dire qu'il représente tous les comportements corrects de l'entité, et précis, c'est-à-dire qu'il ne représente que les comportements corrects de l'entité. Les approches de détection d'intrusions que nous avons proposées sont fondées sur une technique du domaine de la sûreté de fonctionnement, la diversification fonctionnelle, qui ne nécessite pas la construction d'un modèle explicite de l'entité.

Ce chapitre présente les deux approches de détection que nous avons proposées et étudiées au cours de la thèse : la première de type boîte noire et la seconde de type boîte grise. Ce chapitre présente également de manière générale les algorithmes de détection proposés. Nous allons tout d'abord détailler les principes fondamentaux de la détection d'intrusions par diversification fonctionnelle que nous avons mis en évidence. Ensuite, nous décrirons l'architecture du système de détection d'intrusions proposé en mettant en valeur les similarités et les différences entre les deux approches de type boîte noire et de type boîte grise. Ceci permettra de montrer les avantages et les limites à la fois théoriques et pratiques des approches proposées.

2.1 Principes fondamentaux de la détection d'intrusions par diversification de COTS

Les deux approches de détection d'intrusions que nous avons proposées sont fondées sur la programmation N-versions [AC77], une technique de diversification fonctionnelle [Elm72]. Il n'est pas réaliste d'utiliser cette méthode telle quelle, pour des raisons de coût, et nous avons choisi de remplacer les versions qui sont développées spécifiquement, par des COTS (Components-Off-The-Shelf). Nos deux approches de détection reposent sur la comparaison du comportement de ces COTS. Le choix d'utiliser des COTS implique certaines contraintes que nous présentons d'abord. Ensuite, nous présentons l'architecture générale que nous avons utilisée pour nos deux approches, architecture inspirée de celle de la programmation N-versions. Les différences de comportement entre les COTS ne sont pas forcément dues à des intrusions ; nous classifions ces différences de comportement suivant leur cause et proposons une méthode pour masquer celles qui doivent l'être et ainsi éviter des faux positifs. Finalement, nous discutons des impacts sur les performances qu'entraîne l'utilisation de cette architecture et des choix possibles pour les améliorer.

2.1.1 Utilisation de COTS dans un système de type N-versions

Les deux approches de détection d'intrusions que nous avons proposées sont basées sur la programmation N-versions ; elles s'en différencient par l'utilisation de COTS. Ce choix est motivé par le fait que la programmation N-versions [AC77] est une technique coûteuse puisque chaque version doit être développée de manière indépendante par différentes équipes en utilisant des méthodes de développement différentes, des langages différents, etc. Elle est ainsi réservée à des systèmes critiques du point de vue de la sécurité-innocuité. Il n'est pas viable d'un point de vue économique d'utiliser cette technique pour tout type de système.

Dans le domaine des services liés à l'Internet, de nombreux COTS sont disponibles. Il existe, par exemple, de très nombreux serveurs HTTP : Apache, IIS, IPlanet, thttpd, Lighttpd, Xitami, Caudium, Abyss, MyServer, etc. C'est également le cas pour les serveurs FTP ou POP3 ou encore pour les bases de données. Pour les services Internet, il est ainsi possible de remplacer les versions par des COTS dans un système de type N-versions.

Nous avons déjà mis en évidence (voir 1.2.5) trois problèmes posés par l'utilisation de COTS dans un système N-versions :

- leurs spécifications peuvent ne pas être identiques ;
- les points de décision et les données sur lesquelles sont fondées ces décisions doivent être déterminées ;
- les résultats peuvent être non-comparables à cause de problèmes liés à

l'indéterminisme ;
- les COTS n'ont probablement pas été développés de manière à limiter les défaillances de mode commun (partage de code, communication entre les équipes de développement, etc.).

Les deux premiers points peuvent être contournés, en partie, par l'utilisation de COTS respectant une interface standardisée comme des serveurs HTTP ou FTP, par exemple. Les points de décision et les données de comparaison sont alors naturels. Des différences de spécification peuvent tout de même apparaître au niveau de cette interface standardisée. Il est nécessaire de définir précisément l'algorithme de comparaison et les données de comparaison pour obtenir un compromis entre le taux de faux positifs et la couverture de détection.

Ce qui a été dit ci-dessus concerne évidemment l'approche dite boîte noire. Les approches de type boîte grise se confrontent aux différences de spécification et de conception des différents COTS. Plusieurs méthodes ont été proposées pour pouvoir comparer le comportement interne des COTS (voir section 1.3) :
- utiliser le même COTS en le diversifiant grâce à des méthodes automatiques ;
- apprendre les correspondances entre les comportements internes ;
- utiliser une abstraction pour rendre comparable les comportements internes de différents COTS, méthode que nous avons proposée.

Le troisième point peut être pris en partie en compte en mettant en place des mécanismes permettant la synchronisation de l'état des serveurs.

Pour le quatrième point, le projet DOTS [GPSS04] a montré que, pour les bases de données COTS qu'ils ont utilisés et pour les bugs connus, la diversification de COTS est efficace (voir 1.2.5).

Dans le cadre du projet SITAR [WWB01], R. Wang, F. Wang et Byrd ont analysé les vulnérabilités d'IIS et d'Apache. Leur analyse répertorie les vulnérabilités publiées dans la base de vulnérabilités bugtraq de IIS et d'Apache jusqu'en 2001. Nous avons complété cette analyse jusqu'en 2003 et également analysé les vulnérabilités affectant le serveur thttpd (voir tableaux 2.1, 2.2 et 2.3). Les vulnérabilités sont classées suivant leur effet potentiel sur le serveur. L'étude de ces vulnérabilités montre qu'il n'y a aucune vulnérabilité connue commune entre IIS, Apache et thttpd. Cela conforte l'idée qu'il est généralement possible d'utiliser des COTS dans un système N-versions pour la détection d'intrusions et plus généralement la détection d'erreurs.

Il faut bien avoir conscience que cette étude ne démontre pas qu'il n'y a pas de vulnérabilité commune et exploitable de la même manière sur les différents serveurs. Si les résultats de cette analyse avaient montré que de nombreuses vulnérabilités étaient communes, il aurait été possible de soulever des questions quant à l'indépendance des équipes de programmation ou à d'éventuels emprunts de code.

Notre étude n'a pas montré s'il était possible d'exploiter des vulnérabilités différentes dans ces serveurs avec la même requête. Cela aurait représenté un travail très important puisqu'il aurait fallu vérifier que pour chaque couple de vulnérabilités, il est possible ou non d'exploiter ces vulnérabilités grâce à une

même requête. De manière générale, cela est possible. Le nombre de serveurs utilisés dans les architectures de type N-versions dépendant explicitement du nombre de fautes ou d'intrusions que l'on souhaite tolérer, il n'est pas évident de déterminer ce nombre pour détecter et tolérer les intrusions.

	Déni de service	Intégrité des données	Confidentialité des données	Exécution de code	Autres
Validation d'entrée utilisateur		9930	2060 2300 2503 (dir) 5434 5485 5486 6660 9733	5033 5991 629 6659	4431 4437 5847 5884 6029 9804
Dépassement de limites	1821			5993 8911	7723
Validation d'accès	3176 6117	3596 9302	5256 5990 5992 9471 9599		9571
Gestion de conditions exception-nelles	1760 2216 3790 5787 6320 6662 7254 7332 8135 8137 8725 9921		6065 7255 8926		
Configuration			1656 (dir) 4056 4057		
Conception	5816 7725 8138 9826	4358	1728 3169 6939 6943 8134 8707 9874 9933		10212 6661 9829
Concurrence critique		2182			
autres	8226		3009 (dir)		5981

TABLE 2.1 – Vulnérabilités liées à Apache (dir signifie directory traversal)

	Déni de service	Intégrité des données	Confidentialité des données	Exécution de code	Autres
Validation d'entrée utilisateur	1191 522	6759	1081 1084 1193 1488 149 1578 1756 (dir) 1814 4525 4543 968	2023 286	4483 (XSS) 4486 (XSS) 4487 (XSS) 6072 (XSS) 7731 (XSS) 886
Dépassement de limites	1066 192 2674 6070 7733			1911 2252 307 4474 4476 4478 4485 4855 7734	6069
Validation d'accès		1565	167 189 2280 230 2313 529 657		2719
Gestion de conditions exception-nelles	1101 1190 1476 1819 2144 2218 2440 2453 2483 2654 2690 2717 2736 2973 2977 3191 3194 3195 3667 4479 4482 4846 579 5907 7735	191	882		5900 (XSS)
Configuration		1818	1065 4078 4084		
Conception	1642 521		1174 1499 1832 194 195 2074 2110 3159 4235 477 582 978	1912 6071	1595 (XSS) 9313
Concurrence critique			852	501	
autres	193 9660		2909		8244 (XSS)

TABLE 2.2 – Vulnérabilités liées à IIS (dir signifie directory traversal ; XSS cross-site scripting)

	Déni de service	Intégrité des données	Confidentialité des données	Exécution de code	Autres
Validation d'entrée utilisateur			1737 3528		4601 (XSS) 9474 (XSS)
Dépassement de limites	3562			1248 8709 8906	
Validation d'accès			8924		
Configuration					4131

TABLE 2.3 – Vulnérabilités liées à thttpd (XSS signifie cross-site scripting)

2.1.2 Description de l'architecture

2.1.2.1 Architecture et fonctionnement

L'architecture proposée (FIG. 2.1) est inspirée de l'architecture classique utilisée en programmation N-versions. Cette architecture permet à la fois de détecter et de tolérer des intrusions : nous discutons ces propriétés par la suite (voir 2.1.2.2). Elle se compose de deux éléments : un couple proxy/IDS et un ensemble de serveurs. Nous allons tout d'abord détailler le cheminement d'une requête puis décrire plus précisément les différents éléments de l'architecture et enfin discuter certaines propriétés de cette architecture.

Le traitement d'une requête suit le cheminement suivant :

1. le client envoie la requête au proxy. Ce dernier peut éventuellement effectuer certains traitements (filtrage de certaines requêtes, ajouts de données dans la requête, ...) ;
2. le proxy envoie la requête au service exécuté sur chaque serveur ;
3. chaque service effectue le traitement de la requête qui peut éventuellement nécessiter de contacter d'autres machines ;
4. chaque service envoie la réponse à la requête au proxy ;
5. suite à l'exécution de la requête, chaque serveur envoie les éléments de comparaison à l'IDS ;
6. l'IDS compare ces éléments et essaie de dégager une majorité de serveurs ayant eu le même comportement ;
7. l'IDS indique au proxy quelle réponse ce dernier doit envoyer au client ou si le proxy doit envoyer une réponse d'erreur par défaut ;
8. le client reçoit la réponse envoyée par le proxy.

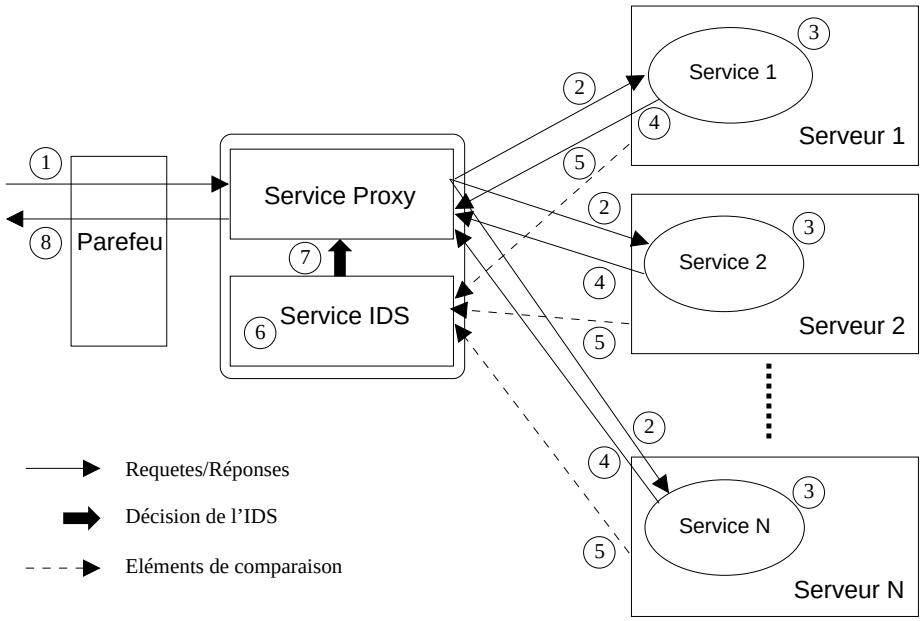


FIGURE 2.1 – Architecture générale

Le proxy/IDS Le proxy/IDS est l'interface avec le client : c'est la seule partie du système visible de l'extérieur. Le proxy est chargé de transmettre les requêtes des clients aux serveurs COTS et ensuite de transmettre en retour au client la réponse que choisit l'IDS. Son rôle est également de synchroniser l'état des serveurs en s'assurant que les serveurs prennent les mêmes décisions d'exécution (ordre de traitement des requêtes, ordre d'ordonnacement des *threads*, etc.). La mise en série des requêtes permet d'assurer une partie de cette synchronisation : c'est ce que nous avons fait dans notre prototype. L'IDS, quant à lui, est chargé de comparer les données choisies lors de la mise en place de l'architecture de type N-versions : ce peut être les réponses des COTS en elles-mêmes (c'est le cas dans l'IDS boîte noire proposé), des séquences d'appels système, des graphes de flux d'informations (c'est le cas dans l'IDS boîte grise proposé). L'IDS est alors chargé de trouver une majorité de serveurs ayant le même comportement du point de vue des éléments comparés (les méthodes de comparaison proposées sont décrites dans les sections 2.2 et 2.3). S'il en trouve une, il indique au proxy de choisir une réponse de cette majorité et lève éventuellement une alerte pour les serveurs ne se trouvant pas dans la majorité. S'il n'y a pas de majorité, l'IDS le signale au proxy qui dans ce cas renvoie un message d'erreur par défaut lié à l'application surveillée et l'IDS lève une alerte.

Les serveurs COTS L'ensemble des serveurs COTS constitue le cœur de l'architecture. Ils fournissent le même service au client. Les serveurs COTS utilisés

doivent être choisis de manière à décorréler au maximum l'activation des fautes. Il est ainsi préférable de choisir des serveurs COTS diversifiés à la fois au niveau des applications utilisées, des systèmes d'exploitation et au niveau matériel (notamment du type de processeurs). Cela permet de minimiser la probabilité qu'il existe des vulnérabilités communes entre les serveurs COTS.

Règles de détection et de localisation La règle générale de détection que nous avons choisie est de lever une alerte dès qu'une différence est détectée dans les observations des comportements des serveurs.

En ce qui concerne la localisation des serveurs compromis, s'il y a une majorité d'observations équivalentes, tous les serveurs dont les observations ne sont pas dans la majorité sont considérés comme compromis. S'il n'y a pas de majorité, nous considérons tous les serveurs comme compromis. C'est la règle de localisation que nous utilisons par la suite.

Exemple de session POP3 Pour rendre plus concret l'approche présentée, nous déroulons un exemple d'une session POP3. Le protocole POP3 (Post Office Protocol, version 3) est un protocole permettant de récupérer les courriers électroniques sur les serveurs de messagerie. Il est possible d'implémenter l'architecture que nous proposons pour ce service. Dans cet exemple, l'IDS compare les séquences d'appels système et l'architecture comprend 3 serveurs. L'un des trois serveurs sera vulnérable à une attaque par débordement de tampons dans la gestion du mot de passe. Nous allons détailler une session qui exploite cette vulnérabilité et une alerte sera levée par l'IDS.

1. Le client se connecte au proxy sur le port TCP 110.
2. Le proxy se connecte aux serveurs COTS sur le port TCP 110.
3. Les serveurs COTS envoient un message de bienvenue au proxy du type `+OK` suivi d'un message :

   ```
   +OK POP3 server ready <7B670F00C11AF35FB0A51C@mail.domain.org>
   ```

   ```
   +OK Hello there
   ```

   ```
   +OK Server ready
   ```

4. Les serveurs envoient à l'IDS la séquence d'appels système exécutés par les services POP3.
5. L'IDS se charge de comparer les séquences d'appels système et ne détecte aucune différence de comportement.
6. Il indique au proxy d'envoyer le message de l'un des serveurs.
7. Le client reçoit la réponse de la part du proxy et envoie ensuite une commande pour indiquer le nom de l'utilisateur :

   ```
   USER prenom.nom@domain.org
   ```

8. Le proxy reçoit cette commande et la transfère aux serveurs.
9. Ceux-ci envoient une réponse du type `+OK` suivi d'un message, par exemple :

```
+OK Password required
```

```
+OK send password
```

```
+OK Password required
```

10. Ils envoient à l'IDS les appels système exécutés lors du traitement de cette commande.
11. L'IDS compare ces séquences d'appels système et ne détecte aucune différence.
12. Il indique au proxy d'envoyer la réponse d'un des serveurs au client.
13. Le client reçoit la réponse et envoie ensuite la commande `PASS` suivi d'un shellcode exploitant la vulnérabilité. Le shellcode télécharge un fichier sur un serveur HTTP distant, exécute ce fichier et répond au client que tout s'est bien passé en envoyant `+OK shellcode executed`.
14. Le proxy reçoit cette commande et la transfère aux serveurs.
15. Les serveurs non-vulnérables envoient les messages suivants :

    ```
    -ERR [AUTH] invalid user or password
    ```

    ```
    -ERR invalid password
    ```

    Le serveur vulnérable envoie la réponse suivante (déterminée par l'attaquant) :

    ```
    +OK shellcode executed
    ```

16. Les serveurs envoient la séquence d'appels système à l'IDS.
17. l'IDS compare ces séquences et détecte que la séquence correspondant au serveur vulnérable est différente des autres qui sont semblables (la séquence d'appels système va contenir au moins les appels correspondant à la création de la socket et la lecture des données sur cette socket). L'IDS lève une alerte concernant le serveur vulnérable et indique au proxy d'envoyer la réponse d'un des deux autres serveurs.
18. Le client reçoit une réponse lui indiquant une erreur d'authentification.

L'intrusion sur le serveur vulnérable est détectée et le serveur vulnérable a pu être localisé. Dans cet exemple, si le pare-feu bloque le téléchargement du fichier, l'intrusion est tolérée par le système.

2.1.2.2 Propriétés de détection et de tolérance aux intrusions

Cette architecture permet de détecter des intrusions et également de les tolérer dans certains cas. Nous allons tout d'abord considérer que les observations sur le comportement de l'entité sont parfaites : ce n'est effectivement jamais le cas. Nous considérons le cas où elles ne le sont pas ensuite. Nous allons reprendre les différentes hypothèses liées à la programmation N-versions, à la diversification de COTS en sûreté de fonctionnement et en sécurité (voir le tableau 2.4).

	Programmation N-versions	Diversification de COTS
tolérance aux fautes	1. $p_n \neq 0$ mais on suppose $p_n = 0$ 2. les versions non fautives se comportent de la même manière 3. les versions fautives se comportent de manière différente les unes des autres ⇒ il faut $f+2$ versions pour tolérer f fautes	1. $p_c \neq 0$? et $p_c > p_n$ mais on suppose $p_c = 0$ 2. les COTS non fautifs peuvent se comporter de manière différente à cause des différences de spécification 3. les COTS fautifs se comportent de manière différente les uns des autres ⇒ on ne peut pas toujours tolérer f fautes avec $f + 2$ COTS à cause de l'hypothèse 2
tolérance aux intrusions	1. $p_n \neq 0$ mais on suppose $p_n = 0$ 2. les versions non compromises se comportent de la même manière 3. les versions compromises peuvent se comporter de la même manière ⇒ il faut $2f + 1$ versions pour tolérer f intrusions	1. $p_c \neq 0$? et $p_c > p_n$ mais on suppose $p_c = 0$ 2. les COTS non compromis peuvent se comporter de manière différente à cause des différences de spécification 3. les COTS compromis peuvent se comporter de la même manière ⇒ on ne peut pas toujours tolérer f intrusions avec $2f + 1$ COTS à cause de l'hypothèse 2

TABLE 2.4 – Hypothèses de tolérance et détermination du nombre de serveurs pour tolérer f fautes ou intrusions ; p_n est la probabilité de fautes corrélées pour la programmation N-versions ; p_c pour la diversification de COTS

On note p_n la probabilité de fautes corrélées pour la programmation N-versions et p_c la probabilité de fautes corrélées pour la diversification de COTS.

Comme précisé précédemment, la diversification est obligatoire : une simple réplication des composants ne permet ni de détecter ni de tolérer les intrusions (de manière générale, certaines fautes peuvent, par contre, être tolérées par la réplication simple [Gra85, Gra86]).

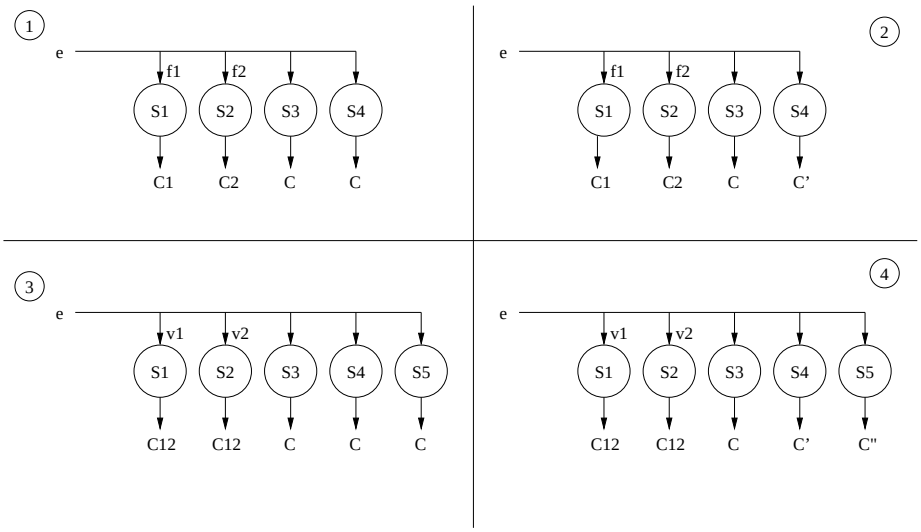


FIGURE 2.2 – Exemples présentant certains cas limites des considérations sur la tolérance aux fautes et aux intrusions dans le cadre de la programmation N-versions et de la diversification de COTS. Dans ces exemples, $f = 2$, e est une entrée ; $f1$, $f2$ des fautes activées ; $v1$, $v2$ des vulnérabilités exploitées ; C les différents comportements.

Programmation N-versions pour la tolérance aux fautes Ce cas correspond à la première case du tableau 2.4. Le processus de développement en programmation N-versions tend à ce que que la probabilité de fautes corrélées soit nulle d'où $p_n \quad 0$. Généralement, on suppose que $p_n = 0$. Dans ce cadre, les fautes sont considérées accidentelles : on en déduit que des fautes décorrélées conduisent à des erreurs distinctes et donc que les versions fautives se comportent de manière différente les unes des autres (ce n'est pas toujours le cas [AK84]). La spécification des versions est la même, donc les serveurs non fautifs se comportent de la même manière. Par un simple vote, il suffit de $f + 2$ versions pour tolérer f fautes : dans une architecture à $f+2$ serveurs, si f fautes sont activées, on peut trouver au moins deux serveurs non-fautifs se comportant de la même manière.

La sous-figure 1 de la figure 2.2 présente le cas $f = 2$. Il est possible de tolérer les fautes activées $f1$ et $f2$ avec une architecture comportant quatre serveurs car les serveurs $S3$ et $S4$ se comportent de la même manière, alors que les serveurs $S1$ et $S2$ se comportent différemment.

Diversification de COTS pour la tolérance aux fautes Ce cas correspond à la deuxième case du tableau 2.4. Une des difficultés liée à l'utilisation de COTS est l'évaluation de la probabilité de fautes corrélées p_c. Elle est certaine-

ment supérieure à p_n car les COTS ne sont pas développés dans un processus de programmation N-versions. L'étude du projet DOTS (voir 1.2.5) montre qu'elle est faible pour certaines bases de données. Généralement, on suppose cette probabilité nulle. Pour les mêmes raisons que celles présentées dans le paragraphe précédent, on fait l'hypothèse que les COTS fautifs se comportent de manière différente les uns des autres. Par contre, à cause des différences de spécification potentielles entre les COTS, les COTS non fautifs peuvent éventuellement se comporter de manière différente. Il n'est alors pas toujours possible de tolérer f fautes avec $f + 2$ COTS. Augmenter le nombre de COTS au-delà de $f + 2$ n'apporte pas la garantie de tolérer f fautes, les spécifications des COTS n'étant en général pas connues.

La sous-figure 2 de la figure 2.2 présente le cas $f = 2$ où l'entrée e active deux fautes $f1$ et $f2$ et les COTS $S3$ et $S4$ ne se comportent pas de la même manière pour cette entrée. Il n'est pas alors possible de tolérer les deux fautes dans ce cas car aucune majorité ne peut être dégagée des comportements des serveurs.

Programmation N-versions pour la tolérance aux intrusions Ce cas correspond à la troisième case du tableau 2.4. L'hypothèse 3 est différente par rapport au cas de la sûreté de fonctionnement. En effet, le modèle de fautes est différent : les fautes considérées sont des fautes intentionnelles. Il n'est plus possible de faire l'hypothèse que des fautes décorrellées conduisent à des erreurs distinctes : l'attaquant peut éventuellement s'assurer que des versions compromises se comportent de la même manière. Ainsi pour tolérer f intrusions, il faut $2f + 1$ versions : la décision, dans ce cas, est basée sur un vote majoritaire. Si f ou moins d'intrusions se produisent sur des versions de l'architecture, au moins $f + 1$ versions vont se comporter de la même manière.

La sous-figure 3 de la figure 2.2 présente le cas où $f = 2$ où l'entrée e exploite deux vulnérabilités $v1$ et $v2$ et conduit les serveurs $S1$ et $S2$ à se comporter de la même manière. S'il n'y avait que quatre serveurs dans l'architecture, il n'aurait pas été possible de tolérer les deux intrusions. Un cinquième serveur permet par vote majoritaire de tolérer les deux intrusions.

Diversification de COTS pour la tolérance aux intrusions Ce cas correspond à la dernière case du tableau 2.4. C'est le cas de notre approche. Il correspond à une combinaison des cases 2 et 3 du tableau. À cause du modèle de fautes et des différences de spécification, il n'est plus assuré de tolérer f intrusions avec $2f + 1$ COTS. De la même manière que pour la tolérance aux fautes, augmenter le nombre de COTS au-delà de $2f + 1$ ne garantit pas de tolérer f intrusions.

La sous-figure 4 de la figure 2.2 est l'analogue de la sous-figure 2 dans le cas des COTS. Si l'entrée e exploite deux vulnérabilités $v1$ et $v2$ et entraîne également des comportements différents parmi les serveurs non compromis, il n'est pas possible de tolérer les intrusions car aucune majorité ne peut être trouvée.

Cas des atteintes à la disponibilité Les atteintes à la disponibilité, autres que les *floods*, visent soit à faire défaillir le serveur ou le service, soit à utiliser de manière disproportionnée les ressources du serveur (processeur, mémoires, ...) pour qu'il ne puisse plus répondre aux requêtes des clients légitimes. Ces défaillances potentielles sont des défaillances temporelles. Il est raisonnable de fixer une limite temporelle à l'envoi des observations (éléments de comparaison) et des réponses. Ces défaillances sont détectées grâce à des chiens de garde (*watchdogs*), au niveau de l'IDS pour les observations et au niveau du proxy pour la réponse du serveur. Si un chien de garde se déclenche, le serveur surveillé par ce chien de garde est considéré comme compromis et doit être reconfiguré. La détection et la localisation sont immédiates dans ce cas.

Détection Nous nous plaçons toujours dans le cas où les observations sont parfaites. Cette hypothèse signifie qu'une intrusion a forcément une conséquence sur les observations et donc que l'on peut considérer que les observations sur le comportement d'un serveur sont le comportement de ce serveur. Si l'on ne cherche qu'à détecter des intrusions et non à les tolérer, il n'est pas nécessaire d'utiliser autant de serveurs. Que ce soit dans le cadre de la programmation N-versions ou grâce à de la diversification de COTS, il faut $f+1$ serveurs pour détecter f intrusions. Si l'on estime qu'il est possible d'exploiter f vulnérabilités grâce à une même entrée, il est nécessaire d'utiliser $f+1$ serveurs pour détecter ces intrusions. En effet, il est nécessaire qu'au moins un des serveurs se comportent de manière différente des serveurs compromis. Ainsi pour déterminer le nombre de serveurs de l'architecture, il est nécessaire d'évaluer la probabilité qu'un attaquant puisse compromettre plusieurs serveurs grâce à une même entrée et faire en sorte que ces serveurs aient le même comportement. Comme expliqué en 2.1.1, cette évaluation n'est pas possible : toutes les vulnérabilités ne sont pas connues, et toutes les attaques exploitant ces vulnérabilités non plus.

La figure 2.3 présente un exemple dans lequel $f = 3$ et l'attaquant est capable de compromettre trois serveurs et faire en sorte que ces serveurs aient le même comportement, la détection reste possible grâce au quatrième serveur. Il faut savoir que l'attaquant n'a pas forcément le contrôle du comportement d'un serveur quand il l'attaque.

Formalisation de la détection Soit N le nombre de serveurs COTS. Soit e une entrée utilisateur qui appartient à l'ensemble des entrées possibles E. On note $C_{i,e}$ le comportement du serveur i pour l'entrée e avec i compris entre 1 et N. La proposition « $C_{i,e}$ est illégal » signifie que le comportement du serveur i pour la requête e a violé la politique de sécurité. La propriété de détection d'une intrusion pour une entrée e s'écrit donc de la manière suivante :

$$\exists l \in \{1, N\} \begin{cases} C_{l,e} \text{ est illégal} \\ \wedge \\ \exists m \in \{1, N\} \setminus \{l\}, C_{m,e} \neq C_{l,e} \end{cases}$$

On note $D_{N,l,e}$ la proposition à droite de l'accolade. La première propriété signifie que l'attaquant arrive à compromettre au moins le serveur l avec l'entrée

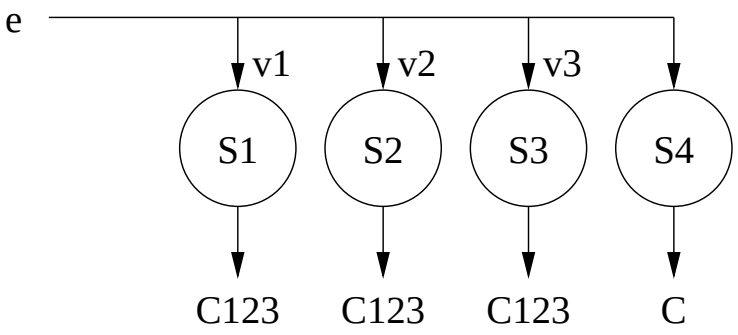


FIGURE 2.3 – Exemple présentant un cas de détection d'intrusions valable dans le cadre de la programmation N-versions et de la diversification de COTS. Dans cet exemple, $f = 3$, e est une entrée ; $v1$, $v2$, $v3$ des vulnérabilités exploitées ; C les différents comportements.

e et la seconde signifie qu'il existe au moins un serveur m dont le comportement est différent de celui du serveur l. Comme les observations sont considérées parfaites, si au moins un serveur n'est pas compromis, son comportement est différent des serveurs compromis.

Si aucun serveur n'est compromis et qu'il existe deux serveurs dont le comportement sont différents, ces différences de comportement proviennent d'une différence de conception ou de spécification, l'alerte émise est un faux positif.

Si aucun serveur ne présente de différences de comportement par rapport à un serveur compromis l, cela signifie que tous les serveurs sont compromis et que l'attaquant a réussi à faire en sorte que le comportement de tous les serveurs soient identiques. Ce cas est un faux négatif. L'architecture est mal dimensionnée par rapport au nombre d'intrusions possibles grâce à une entrée.

Il faut remarquer que la propriété n'indique rien sur le comportement du serveur m, celui-ci peut donc être compromis. Il est donc possible de détecter une intrusion même si tous les serveurs sont compromis dans le cas où ceux-ci se comportent différemment.

Localisation La localisation du ou des serveurs compromis se produit après qu'une alerte a été levée. Cette localisation est plus délicate que la détection. Il faut savoir quels serveurs se sont comportés de manière légale et lesquels se sont comportés de manière illégale. Si on estime le nombre au maximum égal à f, il faut $2f + 1$ serveurs, comme pour la tolérance aux intrusions, si l'on souhaite localiser les serveurs compromis.

Formalisation de la localisation Nous utilisons les mêmes notations que précédemment et introduisons des notations supplémentaires : $C^e = \{C_{i,e} | i \in \{1, N\}\}$, l'ensemble des comportements pour l'entrée e, $C^{M,e}$ le multi-ensemble des comportements identiques formant une majorité absolue ($C^{M,e}$ peut être

vide) et $C^{m,e} = C^e \setminus C^{M,e}$. Si $C^{M,e}$ n'est pas vide, son cardinal est supérieur à $\frac{N}{2}$. On définit la localisation parfaite par la propriété suivante :

$$\exists l \in \{1, N\} \begin{cases} D_{N,l,e} \\ \wedge \\ C^{M,e} \neq \emptyset \\ \wedge \\ \forall i \in \{1, N\},\ C_{i,e} \text{ est illégal} \Rightarrow C_{i,e} \notin C^{M,e} \\ \wedge \\ \forall i \in \{1, N\},\ C_{i,e} \text{ est légal} \Rightarrow C_{i,e} \in C^{M,e} \end{cases}$$

La première propriété est la propriété de détection d'une intrusion sur le serveur l ; la deuxième qu'une majorité a pu être trouvée parmi les comportements de tous les serveurs. Les deux dernières signifient que tous les serveurs compromis ne sont pas dans la majorité contrairement aux serveurs non compromis. Cela signifie qu'aucune différence de spécification n'est intervenue dans le comportement des serveurs non compromis.

Il est possible d'affaiblir la notion de localisation en définissant la localisation imparfaite par la propriété suivante :

$$\exists l \in \{1, N\} \begin{cases} D_{N,l,e} \\ \wedge \\ C^{M,e} \neq \emptyset \\ \wedge \\ \forall i \in \{1, N\},\ C_{i,e} \text{ est illégal} \Rightarrow C_{i,e} \notin C^{M,e} \end{cases}$$

Pour une localisation imparfaite, on n'exige plus que tous les serveurs non compromis soient dans la majorité. Ceci signifie qu'en plus des serveurs compromis, une minorité de serveurs COTS non compromis sont considérés comme compromis par notre approche, ceci à cause de différences de spécification.

Tolérance Comme expliqué dans le tableau 2.4, cette architecture permet, dans certains cas, de tolérer f intrusions dans le système en utilisant $2f + 1$ serveurs COTS. Les intrusions sont tolérées, c'est-à-dire que des composants du système peuvent subir des intrusions mais que celles-ci ne doivent pas entraîner de violations de la politique de sécurité du système global.

La tolérance aux intrusions est assurée par deux mécanismes distincts :

1. vote majoritaire sur les éléments de comparaison permettant de choisir la réponse envoyée au client (c'est-à-dire du masquage d'intrusions) ;
2. détection des intrusions et reconfiguration des serveurs compromis.

Le vote majoritaire permet de tolérer les intrusions contre la confidentialité du système dans le cas où les données confidentielles sont envoyées dans la réponse du serveur compromis. Si le nombre d'intrusions est inférieure ou égal à f et qu'il est possible de trouver une majorité dans le comportement des serveurs COTS, l'intrusion contre la confidentialité est tolérée et une réponse correcte du

point de vue de la spécification est envoyée au client/attaquant. S'il n'est pas possible de trouver une majorité dans les comportements des serveurs, l'intrusion est tout de même tolérée car une erreur par défaut, qui n'est pas forcément correcte du point de vue de la spécification, est envoyée au client/attaquant ; les données confidentielles ne sont pas envoyées au client (dans le cadre où les données confidentielles sont envoyées dans la réponse).

Après une intrusion, les serveurs compromis doivent être retirés de l'architecture. Du code a pu être injecté dans un ou des processus, de nouveaux processus ont pu être lancés, le disque dur a pu être modifié (création ou modification d'un ou plusieurs fichiers), etc. Son fonctionnement correct vis-à-vis de sa spécification n'est ainsi plus garanti. La capacité de tolérance aux intrusions de l'architecture diminue puisque le nombre d'intrusions pouvant être tolérées est égal à $\lfloor \frac{N-1}{2} \rfloor$ où N est le nombre de serveurs de l'architecture. Dans le cas où il n'y a que deux serveurs, sans capacité de diagnostic des intrusions, il n'est pas possible par simple comparaison de localiser le serveur compromis ; les deux serveurs doivent alors être considérés comme compromis. Le service n'est plus rempli par l'architecture. Même si l'objectif final de l'intrusion était une atteinte à la confidentialité ou à l'intégrité, l'intrusion ne peut être considérée comme tolérée car elle a pour effet une indisponibilité (temporaire) du service.

Quand un serveur est retiré de l'architecture, il est possible de le reconfigurer, c'est-à-dire le remettre dans un état considéré comme correct et cohérent par rapport aux autres serveurs de l'architecture. Pour être complet, cette reconfiguration nécessite un redémarrage du serveur et de remettre une image correcte du disque dur sur le serveur, ceci permet d'enlever les traces potentielles de l'intrusion en mémoire et sur le disque dur. Il peut être également nécessaire de rejouer certaines requêtes pour mettre le serveur reconfiguré dans un état cohérent par rapport aux autres serveurs de l'architecture. Dans le cadre du projet BASE [CRL03], un mécanisme de reconfiguration fondé sur des mécanismes de checkpointing et de reprise, a été proposé et décrit en détail. Le mécanisme de détection puis reconfiguration est le dernier mécanisme de tolérance aux intrusions : il permet d'éviter que les effets d'une intrusion se propagent.

Il peut être intéressant, en termes de sécurité, de reconfigurer les serveurs périodiquement car, en cas de faux négatif, un serveur peut être compromis et l'attaquant peut éventuellement rester indétecté tout en violant la politique de sécurité du système.

Il est évidemment possible qu'une intrusion ne soit pas tolérée : d'une part dans le cas où elle n'est pas détectée, d'autre part même si elle est détectée mais que des mécanismes de prévention ne sont pas mis en place ou que le diagnostic de l'alerte n'est pas effectué par l'administrateur de sécurité. Nous allons expliciter ces deux derniers cas.

Supposons qu'un attaquant prenne le contrôle du flot d'exécution d'un des serveurs de l'architecture. Sans un pare-feu très strict, il pourrait envoyer des informations confidentielles en utilisant des canaux de communication permis par le pare-feu sans passer par le proxy : il pourrait faire sortir ces informations

confidentielles en effectuant des requêtes DNS vers un domaine qu'il contrôle.

Si l'administrateur n'effectue pas d'analyse de l'alerte, il est possible qu'une intrusion ne soit pas tolérée même si elle a été détectée. Supposons de la même manière que l'attaquant a réussi à prendre le contrôle du flot d'exécution d'un des serveurs. Il lui est possible d'attaquer un autre composant du système global tel qu'un serveur de base de données ou le serveur DNS et éventuellement de le corrompre. Même si le serveur COTS est lui-même détecté comme compromis et reconfiguré, il faut que l'administrateur puisse diagnostiquer l'intrusion et déterminer qu'un autre composant a été compromis.

Formalisation de la tolérance Il est possible de formaliser les propriétés de tolérance apportées par les deux derniers mécanismes présentés ci-dessus : le vote majoritaire et la détection/reconfiguration.

Nous reprenons les notations introduites précédemment. Nous ajoutons les notations suivantes : $R_{i,e}$ la réponse du serveur i à l'entrée e et $R^{c,e}$ la réponse choisie par le proxy/IDS.

La détection/reconfiguration repose sur les propriétés nécessaires pour obtenir la détection des intrusions et la localisation des serveurs compromis présentées dans la partie concernant la formalisation de la détection et de la localisation.

Le vote majoritaire permet de tolérer les intrusions à la confidentialité en choisissant une réponse « majoritaire » si $C^{M,e}$ n'est pas vide sinon une réponse par défaut. L'attaquant doit ainsi vérifier les propriétés suivantes pour que ce mécanisme de tolérance soit contourné :

$$\exists e \in E, \exists l \in \{1, N\} \begin{cases} C_{l,e} \text{ est illégal} \\ \wedge \\ R_{l,e} = R^{c,e} \end{cases}$$

La seconde propriété : $R_{l,e} = R^{c,e}$ implique que $C^{M,e}$ ne soit pas vide et contienne $C_{l,e}$. Ce qui signifie que l'attaquant a réussi à compromettre plus de la moitié des serveurs COTS. Le nombre de serveurs de l'architecture est insuffisant.

Nous avons étudié et formalisé les mécanismes de détection, de localisation et de tolérance quand les observations sont parfaites, c'est-à-dire qu'une intrusion a forcément une influence sur l'observation. Cela peut ne pas être le cas et c'est une raison pour laquelle nous avons proposé une approche de type boîte grise. Dans la suite, nous développons le cas où les observations sont imparfaites. Nous entendons par observations imparfaites deux concepts distincts mais qui ont la même conséquence. D'une part, une intrusion peut ne pas avoir de conséquence sur les observations, ce qui est un défaut inhérent au choix des éléments de comparaison. D'autre part, l'algorithme de comparaison est imparfait et peut décider que des observations sont semblables bien que l'une des observations corresponde à un comportement intrusif.

Détection d'intrusions : observations imparfaites Nous reprenons les mêmes notations que précédemment. Nous notons $O_{i,e}$ les observations sur le comportement du serveur i pour l'entrée e. Les comparaisons sur les observations sont relatives à l'algorithme de comparaison choisi. Il faut noter qu'un attaquant ne peut pas toujours prendre le contrôle du comportement du serveur et qu'obtenir le contrôle du comportement du serveur n'implique pas forcément obtenir le contrôle des observations sur le comportement du serveur compromis. La détection est réussie pour une entrée e, si la proposition suivante est vraie :

$$\exists l \in \{1, N\} \begin{cases} C_{l,e} \text{ est illégal} \\ \wedge \\ \exists (m, n) \in \{1, N\}^2, O_{m,e} \neq O_{n,e} \end{cases}$$

On note $D^{O}_{N,l,e}$ la proposition à droite de l'accolade. La différence par rapport au cas où les observations sont parfaites (propriété $D_{N,l,e}$) porte sur la seconde propriété. Les observations sur le comportement des serveurs COTS ou l'algorithme de comparaison peuvent ne pas mettre en évidence l'intrusion sur le serveur l.

Il est ainsi possible d'être en présence d'un faux négatif si toutes les observations sont égales bien qu'il y ait eu une intrusion sur au moins l'un des serveurs. Dans le cas où les observations sont parfaites, il est nécessaire pour l'attaquant de réussir une intrusion sur tous les serveurs et de faire en sorte que leur comportement soit identique.

Une alerte est un faux positif s'il existe au moins deux serveurs dont les observations sont jugées différentes par l'algorithme de comparaison alors qu'aucun serveur n'a été attaqué.

mimicry attack Si l'attaquant cherche à masquer son intrusion, il faut qu'il trouve une entrée e vérifiant les propriétés suivantes :

$$\exists e \in E \begin{cases} \exists l \in \{1, N\},\ C_{l,e} \text{ est illégal} \\ \wedge \\ \forall (m, n) \in \{1, N\}^2, O_{m,e} = O_{n,e} \end{cases}$$

C'est le cadre d'une *mimicry attack* : l'attaquant est capable de contrôler le comportement du serveur compromis l et les observations sur son comportement. Il est capable alors de faire en sorte que les observations sur le comportement du serveur compromis l ne soient pas différentes des observations pour les autres serveurs. La difficulté supplémentaire provient du fait que l'attaquant ne maîtrise pas les effets de l'attaque sur les serveurs non compromis qui peuvent réagir de manière différente suivant leur spécification et leur conception. Par exemple pour des serveurs HTTP, une tentative de *buffer overflow* dans l'URL peut être détectée par un serveur non-compromis qui va fermer la connexion alors qu'un autre serveur non-compromis peut chercher le fichier correspondant à l'URL sur le disque et ne va pas le trouver, répondant alors à l'attaquant que le fichier n'existe pas ; les deux serveurs ne sont pas affectés par l'attaque mais les observations sur leur comportement peuvent être différentes.

Localisation : observations imparfaites Nous reprenons les notations précédentes et définissons O^e l'ensemble des observations du comportement de tous les serveurs pour l'entrée e : $O^e = \{O_{i,e} | i \in \{1, N\}\}$; $O^{M,e}$ le multi-ensemble des observations pour l'entrée e qui sont équivalentes du point de vue de l'algorithme de comparaison et qui forment une majorité absolue. $O^{M,e}$ peut être vide. S'il n'est pas vide, son cardinal est supérieur à $\frac{N}{2}$. Nous notons $O^{m,e} = O^e \setminus O^{M,e}$. $O^{m,e}$ contient toutes les observations qui ne sont pas dans la majorité si elle existe. Si la majorité n'existe pas, $O^e = O^{m,e}$. Les observations dans $O^{m,e}$ ne sont pas forcément égales du point de vue de l'algorithme de comparaison.

On dit que la localisation est parfaite pour une entrée e lorsque la proposition suivante est vraie :

$$\exists l \in \{1, N\} \begin{cases} C_{l,e} \text{ est illégal} \\ \wedge \\ \exists m \in \{1, N\}, O_{m,e} \neq O_{l,e} \\ \wedge \\ O^{M,e} \neq \emptyset \\ \wedge \\ \forall i \in \{1, N\}, C_{i,e} \text{ est illégal} \Rightarrow O_{i,e} \notin O^{M,e} \\ \wedge \\ \forall i \in \{1, N\}, C_{i,e} \text{ est légal} \Rightarrow O_{i,e} \in O^{M,e} \end{cases}$$

Les deux premières propriétés représentent la détection de l'intrusion ; la deuxième est légèrement simplifiée à cause des autres propriétés mais cela ne change pas sa signification. La troisième signifie qu'il y a une majorité d'observations équivalentes suivant l'algorithme de comparaison. La quatrième signifie que toutes les observations sur les serveurs compromis ne sont pas la majorité et la cinquième que toutes les observations sur le comportement des serveurs non compromis sont dans la majorité. La localisation parfaite est difficile à assurer à cause des différences de conception/spécification et des observations imparfaites.

De la même manière, il est possible de définir la notion de localisation imparfaite par la proposition suivante :

$$\exists l \in \{1, N\} \begin{cases} C_{l,e} \text{ est illégal} \\ \wedge \\ \exists m \in \{1, N\}, O_{m,e} \neq O_{l,e} \\ \wedge \\ O^{M,e} \neq \emptyset \\ \wedge \\ \forall i \in \{1, N\}, C_{i,e} \text{ est illégal} \Rightarrow O_{i,e} \notin O^{M,e} \end{cases}$$

Certains serveurs non compromis peuvent ne pas être dans la majorité. Ils sont considérés comme compromis par l'IDS. Par contre, tous les serveurs compromis sont considérés comme compromis par l'IDS.

Tolérance aux intrusions : observations imparfaites Le vote majoritaire permet de tolérer les intrusions à la confidentialité en choisissant une réponse

« majoritaire » si $O^{M,e}$ n'est pas vide sinon une réponse par défaut. L'attaquant doit ainsi vérifier les propriétés suivantes pour que ce mécanisme de tolérance soit contourné :

$$\exists e \in E, \exists l \in \{1, N\} \begin{cases} C_{l,e} \text{ est illégal} \\ \wedge \\ R_{l,e} = R^{c,e} \end{cases}$$

Cette proposition est la même que dans le cas où les observations sont parfaites mais son interprétation est différente. La seconde propriété : $R_{l,e} = R^{c,e}$ implique que $O^{M,e}$ ne soit pas vide et contienne $O_{l,e}$. Le choix de la réponse par le proxy/IDS est également un paramètre de l'architecture : plusieurs choix peuvent être faits : choix aléatoire parmi les réponses qui correspondent aux observations qui sont dans $O^{M,e}$, choix de la réponse la plus rapide, détermination d'un ordre sur les serveurs, etc. Dans le cas d'un choix aléatoire, si un seul serveur est compromis, la probabilité que le mécanisme soit contourné, dans le pire des cas, est donc égale à $\frac{1}{\lceil \frac{N+1}{2} \rceil}$ (ceci équivaut à $\frac{2}{N+1}$ si N est impair et à $\frac{2}{N+2}$ si N est pair). Dans le pire des cas, si l'attaquant est capable de compromettre $\lfloor \frac{N-1}{2} \rfloor$, la probabilité que le mécanisme soit contourné est égale à $\frac{\lfloor \frac{N-1}{2} \rfloor}{\lceil \frac{N+1}{2} \rceil}$ (ceci équivaut à $\frac{N-1}{N+1}$ si N est impair et $\frac{N-2}{N+2}$ si N est pair). Ce cas est cependant très peu probable.

Le mécanisme de vote majoritaire est effectif pour la confidentialité lorsqu'une des trois propositions suivantes est vérifiée :

- $O^{M,e} = \emptyset \Leftrightarrow O^{m,e} = O^e$.
- ou $O_{M,e} \neq \emptyset \wedge \forall l \in \{1, N\}$, $C_{l,e}$ est illégal $\Rightarrow O_{l,e} \notin O^{M,e}$.
- ou $O_{M,e} \neq \emptyset \wedge \forall l \in \{1, N\}$, $C_{l,e}$ est illégal $\wedge\ O_{l,e} \in O^{M,e} \Rightarrow R_{l,e} \neq R^{c,e}$.

En ce qui concerne le mécanisme de vote majoritaire, si l'attaquant compromet une majorité de serveurs, il a de grandes chances que la réponse envoyée soit celle qu'il attend. Cette probabilité est de 1 si la majorité n'est constituée que de serveurs compromis, c'est-à-dire $O^{M,e}$ n'est pas vide et ne contient pas les observations des serveurs non compromis.

2.1.3 Classification des différences détectées

La programmation N-versions permet de détecter et tolérer des fautes en comparant la sortie de plusieurs programmes répondant à une même spécification. Les différences détectées correspondent dans ce cas à des fautes de conception dans l'une des variantes puisque la spécification est la même pour toutes les variantes. Les éléments de comparaison (qui peuvent être des états internes à chaque variante) et les points de décision sont précisés dans cette spécification.

L'utilisation de COTS dans une architecture de type N-versions ne permet plus de faire cette hypothèse (Fig. 1.3) : toutes les différences détectées ne sont pas dues à des fautes de conception mais peuvent être liées à des différences de spécification entre les COTS ou à la présence de non-déterminisme.

Dans le cadre de l'approche boîte noire et l'utilisation de COTS respectant une interface standardisée, la spécification d'un COTS peut être décomposée en deux parties : une partie commune qui correspond à la spécification des autres COTS et une partie spécifique qui diffère de celles des autres variantes.

Ainsi, les différences en sorties des COTS (Fig. 2.4) sont la conséquence :

- des différences de conception qui sont dues à des différences dans les parties de la spécification spécifiques à chaque COTS. Ces différences de conception peuvent être dues à des fautes de conception ou non ;
- des différences de conception qui sont dues à des fautes de conception dans la partie de spécification commune entre les différents COTS.
- de non-déterminisme dans l'exécution (threads, informations spécifiques telles que la date, etc.)

Dans le cadre de l'approche boîte grise, la situation est bien plus complexe. Il n'y a aucune raison que le comportement interne des COTS soit directement comparable. La spécification des COTS n'est pas censée imposer tous les choix de conception interne. Certains fonctionnements internes sont cependant dictés par la spécification : de manière générale, un serveur FTP ou HTTP lira le contenu qu'il dessert sur le disque dur, un serveur SMTP redirigera des mails vers d'autres serveurs en ouvrant des connexions réseaux et stockera les mails pour les utilisateurs sur le disque dur. Au niveau de l'interface avec le système d'exploitation, certains comportements des COTS peuvent être communs de manière générale. On peut ainsi établir une figure similaire à la figure 2.4. Ce ne sont plus les sorties qui sont comparées mais une abstraction du comportement des COTS au niveau de l'interface avec le système d'exploitation. De plus, des différences de conception peuvent apparaître même dans la partie où les spécifications des COTS sont communes.

Nous cherchons à détecter des intrusions, c'est-à-dire des différences qui sont la conséquence de l'exploitation d'une vulnérabilité. Ces vulnérabilités sont des fautes de conception. L'exploitation d'une vulnérabilité conduit à une intrusion dans le système, c'est-à-dire une violation de l'intégrité, de la disponibilité ou de la confidentialité. Il est ainsi possible de classer les différences détectées en deux ensembles : d'une part, les intrusions et, d'autre part, les différences de spécification et les fautes de conception classiques, c'est-à-dire les fautes qui ne violent pas la politique de sécurité. Il peut être intéressant de détecter les fautes de conception classiques dans l'optique de les signaler aux développeurs des différents COTS, tout comme les vulnérabilités exploitées. L'administrateur de sécurité doit cependant analyser les différences détectées par l'IDS pour remonter à l'éventuelle faute de conception classique ou vulnérabilité. Notre approche boîte grise permet une aide à ce premier diagnostic. Il faut, par contre, remarquer qu'il n'est pas possible à l'IDS de classer les différences détectées sans l'ajout de diagnostics supplémentaires (expertise humaine, utilisation d'autres IDS, etc.).

Un problème de l'approche réside dans le traitement des différences de spécification qui peuvent conduire à des faux positifs. Plusieurs approches peuvent

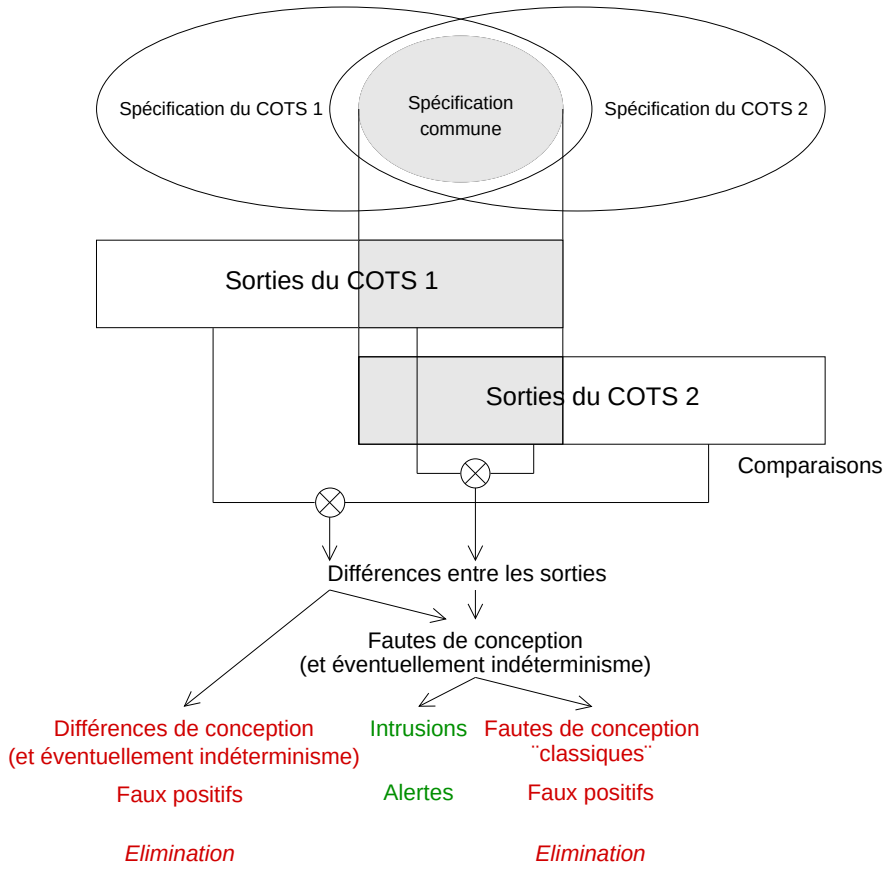


FIGURE 2.4 – Taxonomie des différences en sortie

être utilisées :

- utiliser des algorithmes de comparaison relativement généraux ;
- utiliser un modèle abstrait de la spécification et modifier les réponses grâce à des wrappers pour répondre à ce modèle de spécification ;
- apprendre les correspondances ;
- masquage des différences par apprentissage (manuel ou automatique) ;

La première solution a notamment été utilisée par le projet HACQIT [RJCM03] (voir section 1.3.2.2). Dans le cadre des serveurs web, ils n'utilisaient que le code de retour de la réponse HTTP pour comparer les comportements des deux serveurs utilisés et ne considéraient que trois cas possibles de couple de réponses. Cette méthode permet d'éviter de prendre en compte des différences de spécification minimes et certains comportements non-déterministes entre les serveurs. La détection est moins précise et les auteurs proposent l'ajout d'autres IDS pour compenser ce fait.

La deuxième solution a été proposée dans le cadre du projet BASE [CRL03] (voir section 1.3.2.1). Il propose un modèle abstrait complet de la spécification et utilise des wrappers de conformité pour faire la translation entre les choix de conception de chaque COTS et le modèle abstrait de spécification. Les wrappers implémentent également des fonctions pour gérer les éventuels non-déterminismes qui ne peuvent être masqués par l'abstraction (lecture de la date, par exemple) et les problèmes de concurrence dus aux traitements en parallèle. Cette méthode est difficilement implémentable quand les COTS présentent des comportements très différents ou quand les comportements ne sont pas documentés.

La solution proposée par Gao [GRS06a] (voir section 1.3.3.4) permet de contourner le problème : la méthode repose sur l'apprentissage des correspondances entre les comportements normaux des différents COTS utilisés. La spécification et les différences de spécification n'entrent finalement plus en compte dans cette méthode. Cette méthode a l'avantage d'être générique mais dépend fortement de la base d'apprentissage. Elle peut être affectée éventuellement par deux problèmes : la présence d'attaques dans la base et une couverture partielle de tous les comportements corrects possibles.

La dernière solution que nous décrivons dans la sous-section suivante est celle que nous avons proposée dans cette thèse : elle permet de pallier en partie les défauts des deux premières approches. Elle permet de mettre en place des algorithmes moins généraux pour la détection et ainsi de se passer éventuellement d'IDS supplémentaires et peut s'appliquer à des COTS dont les comportements sont mal documentés ou très différents les uns des autres.

2.1.4 Mécanismes de masquage des différences de spécification

Les mécanismes que nous avons mis en place pour masquer les différences de spécification des COTS et ainsi limiter le nombre de faux positifs ont été implémentés de manière manuelle dans notre prototype de type boîte noire. Il est tout à fait envisageable, en utilisant une base d'apprentissage, de les implémenter de manière automatique.

Pour l'essentiel (et c'est ce que nous avons vérifié expérimentalement), les différences détectées par notre algorithme de comparaison sont dues à des différences de spécification entre les COTS. Les fautes de conception classiques sont très rares dans les COTS car ce sont des produits qui ont souvent été largement testés. De plus, il n'est pas inintéressant de détecter les fautes de ce type, même si ce n'est pas notre objectif et que nous considérons les alertes correspondantes comme des faux positifs.

L'algorithme de comparaison va détecter presque exclusivement des différences entre les observations dues à des différences de spécification et des intrusions. Il s'agit alors d'éliminer les différences entre les observations résultant des différences de spécification puisque ce serait des faux positifs (il n'y a pas eu violation de la politique de sécurité). Cela est réalisé par masquage de deux manières :

– soit par modification de la requête par le proxy ;
– soit par traitement au niveau de l'IDS.

Le mécanisme situé au niveau du proxy est utile lorsque les COTS ne respectent pas tout à fait la même spécification au niveau de leur interface. Il permet de modifier une requête du client avant sa transmission à tous les serveurs ou seulement à certains d'entre eux, la requête originale étant alors envoyée aux autres serveurs. Par exemple, certains serveurs HTTP refusent de traiter une requête contenant une remontée dans l'arborescence même si celle-ci ne fait pas sortir de l'arborescence desservie par le serveur. Dans ce cas, il est possible par exemple de modifier la requête en simplifiant le chemin indiqué par l'URL.

Le mécanisme présent au niveau de l'IDS permet de masquer des différences détectées qui sont des différences de spécification connues afin de ne pas lever d'alertes pour ces différences. Des règles de masquage sont alors définies. Une règle établit une relation entre une classe d'entrées et des classes de sorties de chaque serveur COTS et associe à cette relation un ensemble d'actions. Les actions permettent la modification de certaines parties des réponses des serveurs. Ces modifications permettent de rendre les réponses identiques du point de vue de l'algorithme de détection et ainsi de masquer les différences de spécification entre les serveurs COTS.

Si N est le nombre de serveurs utilisés dans l'architecture. Une règle est un $(N+2)$-uplet $(E, S_1, S_2, \ldots, S_N, A)$ où E est un sous-ensemble des entrées possibles, S_i un sous-ensemble de sorties du serveur i et A un ensemble d'actions. Lorsqu'un ensemble de réponses $\{s_1, s_2, \ldots, s_N\}$ à une entrée e est reçu, si e fait partie de E et si, pour tout i, s_i fait partie de S_i, alors les actions de l'ensemble A sont effectuées.

Les règles de masquage telles que nous les avons définies peuvent éventuellement masquer une intrusion si l'attaquant est capable d'envoyer une requête qui induit des réponses qui correspondent à une règle. L'ensemble d'actions pourrait masquer les différences de comportement imputables à l'intrusion sur le serveur compromis. La définition de ces règles de masquage doit donc être suffisamment précise pour ne pas introduire de faux négatifs : une différence due à l'exploitation d'une vulnérabilité ne doit pas être masquée par ces règles. Ainsi la pertinence de la détection dépend de l'ensemble des règles mises en place par l'administrateur et de leur qualité.

Les règles présentes au niveau du proxy et de l'IDS constituent un modèle explicite des différences entre les serveurs. Ce modèle est relativement réduit et facile à construire car on ne modélise que les différences entre les serveurs COTS et non les serveurs COTS complets. La nécessité d'introduire ces règles montrent que le modèle implicite que constitue chaque COTS pour les autres n'est pas complet ni précis dans le cas général.

Pour établir ce modèle, la solution la plus simple est d'identifier expérimentalement ces différences et de développer les règles et mécanismes pour les masquer à un des deux niveaux décrits. Ces règles sont dépendantes des COTS utilisés et

de leur version. Une nouvelle version peut apporter de nouvelles fonctionnalités, corriger des bugs. Les règles doivent donc être mises à jour, si nécessaire, lors d'un changement de version d'un des serveurs COTS.

Ces mécanismes ont pour but de réduire le nombre de faux positifs qu'engendrerait un algorithme de comparaison seul. Réduire le nombre de faux positifs est un problème essentiel en détection d'intrusions, surtout pour les approches comportementales, et il est nécessaire de les limiter au maximum pour qu'une approche soit utilisable en pratique [Axe00a]. Dans le cadre de notre approche, cela est encore plus essentiel car l'IDS est très intimement lié à l'architecture. Les faux positifs et les faux négatifs ont un impact important sur les capacités de détection.

2.1.5 Influence des faux négatifs et des vrais et faux positifs

Les faux négatifs et les vrais et faux positifs vont affecter les performances de l'IDS mais également les performances globales de l'architecture que nous étudierons plus complètement dans la sous-section suivante 2.1.6.

2.1.5.1 Effets des vrais et faux positifs

Les effets des vrais positifs et des faux positifs sont les mêmes sur l'architecture, excepté le fait que les mesures qui doivent être prises en cas d'alertes ne sont pas nécessaires en cas de faux positifs alors qu'elles le sont en cas de vrais positifs.

Il faut distinguer deux cas : la localisation du ou des serveurs éventuellement compromis a été possible et le cas contraire où la localisation du ou des serveurs éventuellement compromis ne l'a pas été.

Dans le premier cas, notons M le nombre de serveurs considérés comme compromis. Comme la localisation a « réussi », on a : $0 < M < \frac{N}{2}$. Ces M serveurs considérés comme compromis doivent être retirés de l'architecture et reconfigurés. L'architecture comprend alors $N \quad M$ serveurs : les capacités de détection et de tolérance aux intrusions de l'architecture diminuent. Si l'architecture n'a plus que deux serveurs, le vote majoritaire n'est plus possible en cas de différences dans les observations des comportements des deux serveurs restants. Si les observations sont différentes, les deux serveurs sont considérés comme compromis et le service est alors indisponible.

Un attaquant peut profiter de ce fait de deux manières. Il peut développer une attaque contre la disponibilité du service par une suite de requêtes entraînant des différences dans les observations des comportements de serveurs COTS, obligeant à retirer tous les serveurs de l'architecture et les reconfigurer. Il peut également diminuer volontairement les capacités de détection et de tolérance de l'architecture par des requêtes entraînant des faux positifs puis tenter une intrusion qui ne sera pas détectée car le nombre de serveurs ne sera pas suffisant. Dans ce second scénario, son attaque n'est cependant pas discrète car ses

premières requêtes génèrent des alertes.

Si l'alerte est un vrai positif mais que la localisation a échoué dans le sens où le serveur réellement compromis ne fait pas partie des serveurs considérés comme tels, ce serveur n'est pas retiré de l'architecture, ni reconfiguré. La requête est détectée comme intrusive mais aucune mesure automatique n'est prise pour limiter les effets de l'intrusion. Ce cas correspond presque au cas d'un faux négatif à part le fait qu'une alerte a été générée et que l'administrateur de sécurité pourra procéder à une analyse de l'alerte.

Dans le second cas, une seule requête suffit à créer une indisponibilité du service : tous les serveurs doivent en effet être reconfigurés. Si l'attaquant est capable de trouver une requête ayant pour conséquence l'impossibilité de localiser les serveurs éventuellement compromis, il peut rendre le service indisponible.

Il est donc nécessaire de limiter fortement les faux positifs car, dans ce cas, les reconfigurations sont inutiles et diminuent les capacités de détection et de tolérance. Il faut également améliorer la localisation des serveurs compromis si l'on veut éviter des attaques contre la disponibilité du service.

2.1.5.2 Effets des faux négatifs

Les faux négatifs ont un effet potentiellement important sur l'IDS du fait de la liaison très forte entre le service et l'IDS. Si l'attaquant est capable de contrôler le comportement et les observations du serveur qu'il a compromis, il peut tenter de compromettre successivement les autres serveurs et, à terme, obtenir une majorité de serveurs sous son contrôle. Dans ce cas, les capacités de détection et de tolérance de l'architecture sont fortement limitées : les serveurs éventuellement considérés comme compromis seront dans les faits les serveurs non compromis, l'attaquant pourra contourner le vote majoritaire pour accéder à des informations confidentielles, etc.

La seule manière de limiter le problème des faux négatifs est de reconfigurer de manière périodique tous les serveurs de l'architecture.

2.1.6 Performances de l'architecture

Tout mécanisme de sécurité a généralement un coût en termes de performances sur le service. Nous étudions dans cette sous-section quels sont les éléments qui vont dégrader les performances du service et les solutions éventuelles pour limiter cette dégradation. Ces solutions ont, par contre, un contrecoup en termes de sécurité.

2.1.6.1 Obstacles aux performances

L'architecture décrite précédemment permet à la fois de détecter des intrusions et de tolérer des intrusions au prix d'un certain coût en performance : le fonctionnement de l'architecture tel que présenté section 2.1.2.1 entraîne une augmentation du temps de réponse et la sérialisation du traitement des requêtes

afin de parer aux problèmes de non-déterminisme liés à leur exécution concurrente peut entraîner une baisse du débit des requêtes traitées.

Latence Tel que nous avons présenté le fonctionnement de l'architecture, le proxy attend la décision de l'IDS pour renvoyer la réponse au client. L'architecture introduit donc une latence à plusieurs endroits du traitement de la requête.

Dans le cas d'un serveur seul répondant aux requêtes des clients, sans mécanisme de sécurité supplémentaire, le temps de réponse va être égal à la somme du temps d'envoi de la requête, du temps de traitement au niveau du serveur et du temps d'envoi de la réponse. On note respectivement $t_{rep,s}$, $t_{req,c\to s}$, t_{tr} et $t_{rep,s\to c}$ les quatre temps précédents. On a donc :

$$t_{rep,s} = t_{req,c\to s} + t_{tr} + t_{rep,s\to c}$$

L'architecture introduit de nombreux délais supplémentaires dans le traitement d'une requête. On note t_{pr} le temps de traitement de la requête au niveau du proxy, $t_{req,pr\to i}$ le temps d'envoi la requête du proxy au serveur i, $t_{tr,i}$ le temps de traitement de la requête au niveau du serveur i, $t_{rep,i\to pr}$ le temps d'envoi des réponses du serveur i au proxy. Il faut également prendre en compte les temps liés à l'IDS : on note $t_{obs,i}$ le temps nécessaire pour collecter les observations sur le comportement du serveur i, $t_{obs,i\to ids}$ le temps d'envoi des données d'observation du serveur i à l'IDS, t_{ids} le temps de traitement au niveau de l'IDS, $t_{dec,ids\to pr}$ le temps d'envoi de la décision de l'IDS au proxy. De la même manière que ci-dessus, on note $t_{req,c\to pr}$ le temps d'envoi de la requête du client au proxy et $t_{rep,pr\to c}$ le temps d'envoi de la réponse du proxy au client. Le temps de réponse va être égal à :

$$t_{rep,a} = t_{req,c\to pr} + t_{pr} + t_{max} + t_{ids} + t_{dec,ids\to pr} + t_{rep,pr\to c}$$

où $t_{max} = max(\{t_{req,pr\to i} + t_{tr,i} + t_{rep,i\to pr} | i \in \{1, N\}\} \cup \{t_{req,pr\to i} + t_{obs,i} + t_{obs,i\to ids} | i \in \{1, N\}\})$. t_{max} représente le temps que prend le serveur le plus lent pour soit recevoir, traiter la requête et envoyer la réponse, soit pour recevoir la requête, collecter les observations et les envoyer au proxy.

Il est possible de simplifier cette expression en faisant certaines hypothèses. Même si le temps d'envoi de la requête du proxy aux serveurs est différent, ces temps sont sensiblement les mêmes ; on peut donc noter $t_{req,pr\to s}$ $t_{req,pr\to 1}$ $t_{req,pr\to N}$. De manière générale, on peut considérer que le temps de collecte des observations sera supérieur au temps de traitement de la requête : $t_{obs,i}$ $t_{tr,i}$. L'expression de t_{max} devient alors : $t_{max} = t_{req,pr\to s} + max(\{t_{tr,i} + t_{rep,i\to pr} | i \in \{1, N\}\} \cup \{t_{obs,i} + t_{obs,i\to ids} | i \in \{1, N\}\})$.

Dans le cas de notre approche boîte noire, les observations considérées sont les réponses des serveurs. Ainsi, on obtient : $t_{obs,i} + t_{obs,i\to ids}$ $t_{tr,i} + t_{rep,i\to pr}$. Dans le cadre de notre approche boîte grise, on a : $t_{obs,i}$ $t_{tr,i} + t_{rep,i\to pr}$. Dans les deux cas, on peut donc simplifier l'expression de t_{max} : $t_{max} = t_{req,pr\to s} + max(\{t_{obs,i} + t_{obs,i\to ids} | i \in \{1, N\}\})$.

Si on considère de plus comme négligeables les temps de transfert sur le réseau local (ce qui n'est pas forcément vrai), l'expression de t_{rep} peut être simplifiée en :

$$t_{rep,a} = t_{req,c \rightarrow pr} + t_{pr} + max(\{t_{obs,i} | i \in \{1, N\}) + t_{ids} + t_{rep,pr \rightarrow c}$$

En comparant les temps de réponses d'un serveur $t_{rep,s}$ et de l'architecture $t_{rep,a}$, une latence supplémentaire est ajoutée par le traitement de la requête dans le proxy, le traitement des réponses dans l'IDS et à la récupération des observations sur le serveur le plus lent.

Débit de traitement des requêtes La sérialisation des requêtes afin de parer à certains problèmes de non-déterminisme liés à leur exécution concurrente peut entraîner une diminution des performances de l'architecture.

De manière générale, les serveurs COTS tels que les serveurs HTTP, FTP, POP3, par exemple, sont conçus de manière à pouvoir traiter les requêtes en parallèle, grâce à l'emploi de *threads* et/ou de plusieurs processus. Si l'on veut que les comparaisons des comportements soient cohérentes, il est nécessaire que les états des différents serveurs COTS soient synchronisés. Si ce n'est pas le cas, cela peut conduire à des faux positifs.

Deux cas peuvent se présenter :

- les requêtes des clients sont indépendantes entre elles, c'est-à-dire le comportement des serveurs ne dépend pas de l'ordre d'arrivée et de traitement des requêtes ;
- les requêtes des clients peuvent avoir des dépendances entre elles, c'est-à-dire le comportement des serveurs peut dépendre de l'ordre d'arrivée et de traitement des requêtes. Ces dépendances sont la conséquence d'opérations en écriture sur le système considéré.

Le premier cas ne nécessite finalement pas de synchronisation de l'état entre les serveurs. Les requêtes peuvent être traitées de manière concurrente par l'architecture. C'est le cas typique d'un serveur web statique par exemple ou d'un serveur FTP qui n'autorise pas les clients à déposer des fichiers.

Le second cas nécessite une synchronisation du traitement des requêtes par les serveurs. C'est le cas, par exemple, d'un serveur web dynamique ou d'un serveur FTP permettant le dépôt de fichiers. Cette synchronisation est assurée par le proxy en donnant un numéro de séquence à chaque requête.

La solution la plus simple pour assurer la synchronisation est d'ordonner toutes les requêtes en fonction de leur ordre d'arrivée au niveau du proxy. Le proxy envoie les requêtes en série. Ceci peut cependant dégrader fortement les performances de l'architecture comparée à un serveur seul.

Une solution plus élégante est de calculer les dépendances entre les requêtes et de ne mettre en série que les requêtes ayant des dépendances entre elles. Suivant le service considéré, cette solution est plus ou moins difficile. Cela est relativement évident pour un système de fichier NFS [CRL03] mais peut être beaucoup plus complexe pour des requêtes HTTP ou des requêtes SQL. Cette

seconde solution permet de limiter l'influence de la mise en série et d'exécuter de manière concurrente toutes les requêtes indépendantes au prix d'un traitement un peu plus conséquent au niveau du proxy. Nous n'avons pas exploré plus avant cette solution.

2.1.6.2 Compromis performances-sécurité

Des solutions sont envisageables pour limiter les dégradations des performances entraînées par les deux problèmes cités ci-dessus. Ces solutions ont par contre une influence sur les propriétés de sécurité offertes par l'architecture.

Compromis pour pallier le problème de latence Il est possible de modifier le fonctionnement de l'architecture de manière à limiter (voire améliorer) la latence introduite par l'architecture.

Le proxy peut envoyer une réponse au client sans attendre la décision de l'IDS. Plusieurs stratégies peuvent être développées pour le choix de la réponse à envoyer au client. Un ordre sur les serveurs peut être défini, par exemple sur la base d'une évaluation de la sécurité de chaque serveur. Le proxy envoie alors la réponse du premier serveur dans la liste établie si celui-ci répond. Le serveur dont la réponse est envoyée peut être choisi aléatoirement. La réponse envoyée au client est la première réponse reçue par le proxy.

En utilisant les mêmes notations qu'auparavant, le temps de réponse devient donc :

$$t_{rep,a} = t_{req,c\rightarrow pr} + t_{pr} + t_{serveur} + t_{rep,pr\rightarrow c}$$

où $t_{serveur} = t_{req,pr\rightarrow i} + t_{tr,i} + t_{rep,i\rightarrow pr}$ va être le temps de réponse d'un serveur déterminé (grâce à l'ordre défini ou choisi aléatoirement) ou $t_{serveur} = min(\{t_{req,pr\rightarrow i} + t_{tr,i} + t_{rep,i\rightarrow pr} | i \in \{1, N\}\})$ si la réponse choisie est la première reçue.

Les latences introduites alors par l'architecture sont liées au temps de transfert sur le réseau local et au temps de traitement au niveau du proxy. Si ces temps sont relativement négligeables par rapport au temps de traitement par un serveur, il est même possible d'obtenir de meilleures performances qu'en utilisant un seul serveur notamment si le choix du proxy est d'envoyer la première réponse reçue.

Cette solution contourne un des mécanismes de tolérance aux intrusions : le vote majoritaire. Les atteintes à la confidentialité ne sont donc éventuellement plus tolérées puisque l'attaquant peut recevoir une réponse comprenant des informations confidentielles.

Compromis pour pallier le problème de débit Pour éviter que la mise en série des requêtes ait un impact du point de vue des utilisateurs, il est également possible de modifier l'architecture en désignant un serveur leader (voir Fig. 2.5). Le proxy envoie toutes les requêtes au leader sans les mettre en série et

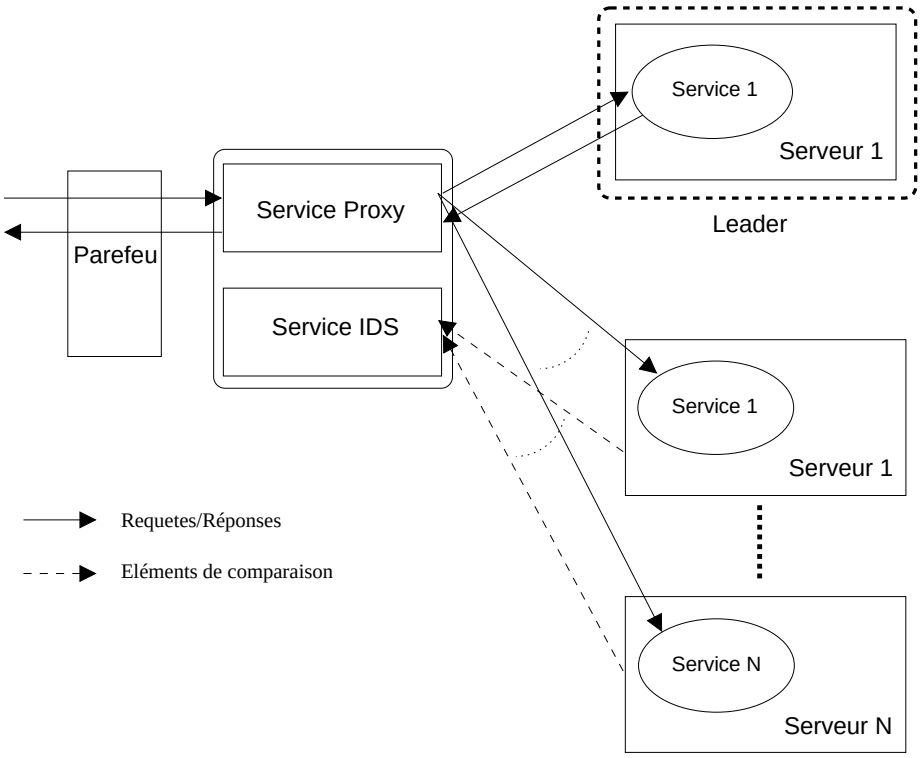


FIGURE 2.5 – Architecture générale modifiée pour limiter la dégradation de performances au niveau du débit de requêtes traitées

renvoie les réponses du leader au client. Ainsi le leader peut traiter les requêtes de manière concurrente et assurer un débit plus important. Le proxy envoie également les requêtes mais mises en série aux autres serveurs sans prendre en compte leurs réponses. Les observations sur leur comportement sont envoyées à l'IDS qui les compare et décide de lever une alerte ou non. Il est raisonnable d'utiliser un serveur COTS identique en tout point (processeur, système d'exploitation, logiciels, ...) au leader parmi les serveurs dont le comportement sera comparé.

Ce choix d'architecture va avoir impact relativement important sur les propriétés de sécurité.

En terme de détection, il va surtout être intéressant de détecter les intrusions qui affectent le serveur COTS qui est la réplique du leader puisque c'est ce dernier qui rend le service au client.

Premièrement, la détection ne se fait plus en temps réel mais si le débit de requêtes n'est pas trop important en quasi temps-réel. Si le débit de requêtes est

élevé, la partie détection de l'architecture peut prendre du retard par rapport au leader.

Le vote majoritaire ne fait plus partie du chemin de traitement des requêtes. Ce mécanisme de tolérance ne fait plus alors office. De même que dans le paragraphe ci-dessus, certaines atteintes à la confidentialité contre le leader ne seront plus tolérées.

Certaines intrusions contre le leader fondées sur des *race conditions* peuvent ne pas être détectées de par le fait que l'ordre des requêtes n'est pas le même pour la réplique que pour le leader. Le leader peut donc être compromis sans que cela ne soit détecté par l'IDS. Il est préférable d'ajouter d'autres IDS pour se prémunir contre ce type d'intrusions.

Si une intrusion contre la réplique est détectée, il est probable que le serveur leader a également été affecté par l'intrusion. Il est nécessaire de le reconfigurer, tout comme la réplique. Plusieurs solutions sont possibles pour le remplacer : choisir temporairement un autre leader parmi les (N 1) serveurs COTS restant, sachant qu'il n'y aura pas de serveur réplique ou revenir temporairement à l'architecture proposée initialement dans la section 2.1.2.1.

2.1.7 Extension de l'architecture à la tolérance aux défaillances du proxy

Dans le cadre de l'ACI-SI DADDi (*Dependable Anomaly Detection with Diagnosis*), en collaboration avec l'IRISA, nous avons étendu l'architecture dans le but de tolérer les défaillances franches du couple proxy/IDS [HNM+06]. L'architecture que nous avons proposée présente, en effet, un seul point de défaillance dur au niveau du proxy/IDS. Dans le but d'améliorer la disponibilité de ce composant, nous proposons l'utilisation de techniques classiques en sûreté de fonctionnement : la réplication et les services d'accord. Les services d'accord sont fournis par une bibliothèque de composants d'accord EDEN [Tro03]. Cette bibliothèque est développée à l'IRISA et est utilisée pour fournir des services d'accord dans les systèmes distribués asynchrones où des défaillances franches sont possibles. Cette approche a été réutilisée dans le cadre des serveurs web pour assurer l'intégrité des requêtes SQL grâce à des proxys SQL formant un groupe d'accord. La première partie de l'architecture (réplication des proxys/IDS) a fait l'objet d'une implémentation commune entre l'IRISA et Supélec.

Nous avons présenté la partie commune entre les deux approches que nous avons proposées. Dans la suite, nous allons détailler les parties spécifiques aux deux approches puis les comparer.

2.2 Approche de type boîte noire : comparaison des sorties réseau

L'approche de type boîte noire que nous avons proposée consiste en la comparaison des sorties réseau au niveau d'une interface standardisée comme par exemple les protocoles HTTP, FTP ou POP3. Cette approche a fait l'objet de deux publications internationales [TMM05, MTM07] et d'une publication nationale [MTM05]. Les observations récupérées et comparées par l'IDS sont donc les réponses des COTS aux requêtes. C'est une approche naturelle comparée à l'approche boîte grise car tous les COTS doivent se conformer à la spécification du protocole. L'approche boîte noire est très proche du modèle général défini précédemment. Nous allons insister sur certaines spécificités : la méthode de comparaison et le masquage des différences.

2.2.1 Méthode de comparaison

Nous avons voulu évaluer les possibilités en terme de détection d'une approche fondée sur la diversification fonctionnelle. Nous souhaitions fonder l'IDS seulement sur la diversification fonctionnelle et ne pas ajouter d'autres IDS réseau ou hôte comme cela a pu être fait dans le cadre d'autres projets tels que DIT [VAC+02] ou HACQIT [RJCM03].

Notre algorithme de comparaison doit être alors suffisamment précis pour détecter la plupart des intrusions mais en limitant le nombre de faux positifs potentiels. Une approche comme celle du projet BASE [CRL03] n'est pas réellement applicable aux serveurs réseaux car les différences de comportement sont parfois réellement importantes entre les COTS et il n'est pas possible d'établir une spécification abstraite valable pour tous les COTS.

La spécification des protocoles va définir différentes parties ayant une sémantique précise dans les réponses au niveau réseau. Nous proposons de comparer certains éléments spécifiques et significatifs de la spécification des différents protocoles. Certains codes de réponses correspondent à des cas précis et ces codes peuvent être comparés de manière évidente. D'autres éléments non-déterministes (valeurs aléatoires, par exemple) peuvent éventuellement être comparés entre eux ou par rapport à un modèle créé auparavant.

Pour reprendre l'exemple du protocole POP3 présenté précédemment dans ce chapitre, les messages des serveurs se composent d'une réponse `+OK` ou `-ERR` suivi d'un message textuel. La première partie de la réponse comporte le sens principal de la réponse : le traitement correct ou incorrect de la requête. Le message textuel n'est pas précisé dans la spécification du protocole : ces messages peuvent donc être entièrement différents et donc difficilement comparables. Il s'agit d'une partie non-spécifiée du protocole. Plusieurs solutions peuvent être envisagées : ne pas les comparer, utiliser une méthode de comparaison approximative (dans l'exemple de POP3, une comparaison approximative des chaînes de caractère [Nav01]) ou comparer ces éléments avec un modèle précédemment appris.

Toutes ces solutions apportent un compromis différent en termes de couverture de détection et de possibles faux positifs et faux négatifs. L'absence de comparaison limite le nombre de faux positifs mais permet éventuellement à l'attaquant de se servir des éléments non-comparés comme des canaux cachés du point de vue de l'algorithme de comparaison pour porter atteinte à la confidentialité du système. Les comparaisons approximatives ne résolvent pas vraiment réellement ce problème mais rendent la tâche de l'attaquant éventuellement plus ardue, tout en laissant présager un nombre de faux positifs accru. Les comparaisons par rapport à un modèle précédemment appris peuvent limiter les possibilités de l'attaquant mais nécessitent une phase d'apprentissage et introduit donc les problèmes liés à l'apprentissage : couverture et généralisation de l'apprentissage, base d'apprentissage saine, etc.

Dans notre prototype de type boîte noire, nous avons choisi de ne pas comparer ces éléments non-spécifiés mais il serait intéressant d'introduire des éléments d'apprentissage car le découpage en *token* est intéressant a priori dans le cadre des réponses réseau comme le montrent Kruegel et al. [KTK02] ou plus spécifiquement pour le protocole HTTP Estévez-Tapiador [ETGTDV04] ou Ingham et Inoue [II07].

Notre algorithme de détection doit être précis pour compenser l'absence d'autres IDS ; il est alors plus susceptible de générer des faux positifs. Nous l'avons donc couplé à un mécanisme de masquage des différences de spécification pour limiter le nombre de faux positifs.

2.2.2 Masquage des différences de spécification

Le mécanisme de masquage de différences s'intègre au modèle présenté précédemment en 2.1.4. Certaines requêtes doivent être modifiées au niveau du proxy et certaines réponses doivent parfois être modifiées pour masquer des différences de spécification.

Il est parfois nécessaire de masquer certaines différences liées au non-déterminisme comme la prise en compte des *cookies* (quand ceux-ci sont générés de manière aléatoire) dans le protocole HTTP ou de l'UID (identificateur unique) dans le cas du protocole POP3. Sur le principe, notre proposition a exactement le même effet que celle du projet BASE [CRL03]. Dans cet article, les auteurs proposent de masquer ces différences par une spécification abstraite vérifiée au niveau des wrappers de conformité. Nous avons choisi de ne pas définir de spécification abstraite mais de prendre l'une des valeurs envoyées par un serveur comme valeur de référence : le proxy se charge ensuite de modifier la requête envoyée par le client pour chaque serveur.

Dans la suite, nous présentons le principe de détection de notre approche de type boîte grise qui comporte plus de spécificités par rapport au modèle global de détection que notre approche boîte noire.

2.3 Approche de type boîte grise : graphes de flux d'informations

Les approches de type boîte noire ont globalement deux défauts : un taux de couverture qui se limite aux intrusions ayant un effet sur la sortie considérée et un défaut commun à beaucoup d'IDS comportementaux, une absence de diagnostic des anomalies détectées. L'approche de type boîte grise que nous proposons tente de résoudre en partie ces deux problèmes. Elle repose sur le calcul d'une similarité entre les graphes de flux d'informations des différents COTS. Les observations sur le comportement des serveurs récupérées et comparées par l'IDS sont des graphes de flux d'informations. Cette approche a fait l'objet d'une publication internationale [MTMS08] et d'une publication nationale [MTMS07].

Nous allons d'abord définir les notions de flux d'informations et de graphes de flux d'informations, puis nous présenterons le modèle et les algorithmes de calcul de la similarité entre les graphes de flux d'informations. Enfin, nous montrons comment il est possible de détecter des intrusions en utilisant ce calcul de similarité.

2.3.1 Notions de flux d'informations

Il est nécessaire de définir la notion de flux d'informations. Nous considérons qu'un flux d'informations d'un objet o_1 vers un objet o_2 a eu lieu si l'état de l'objet o_2 dépend causalement [d'A94] de l'état de l'objet o_1.

Bell et LaPadula [BP76] ont modélisé la création de flux d'informations. Ils ont pour cela considéré deux types d'objets dans le système : les objets actifs (tels que les processus, les utilisateurs, etc. suivant le niveau de granularité considéré) et les objets passifs qui sont les conteneurs d'informations (tels que les fichiers, les sockets, les pipes, les segments mémoires, les variables, etc. suivant le niveau de granularité considéré). Les flux d'informations peuvent être produits par les trois opérations suivantes :

- Un objet actif peut lire un objet passif, ce qui produit un flux d'informations de l'objet passif vers l'objet actif. Un exemple de ce type d'opérations est la lecture d'un fichier par un processus.
- Un objet actif peut écrire un objet passif, ce qui produit un flux d'informations de l'objet actif vers l'objet passif. Un exemple de ce type d'opérations est l'écriture d'informations dans un fichier par un processus.
- Un objet actif o_1 peut invoquer un objet actif o_2, ce qui produit un flux d'informations de o_1 vers o_2. Un exemple de ce type d'opérations est la création d'un nouveau processus par un autre processus.

Dans notre approche, nous nous intéressons aux flux d'informations au niveau du système d'exploitation. Les objets actifs et passifs considérés seront donc des processus, des fichiers, des sockets, des pipes, etc.

Nous avons considéré les flux d'informations au niveau du système d'exploitation pour deux raisons.

Cela permet d'avoir une granularité suffisamment importante pour masquer, en partie, les différences de conception entre les COTS. Notre hypothèse, qui est vérifiée pour notre prototype d'IDS pour les serveurs web, est que les flux d'informations, à ce niveau de granularité et en l'absence d'intrusions, sont très semblables dans les COTS. Nous pensons que cette hypothèse se vérifie pour les serveurs HTTP, FTP, POP3, SMTP par exemple. Il est beaucoup plus difficile d'émettre cette hypothèse en considérant les flux d'informations à l'intérieur de l'application COTS elle-même : par exemple, les paramètres et appels des fonctions ont très peu de chances d'être comparables entre les différents COTS.

L'autre raison est qu'une intrusion, en général, aura une manifestation au niveau des flux d'informations au niveau du système d'exploitation. En effet, une intrusion contre la confidentialité ou l'intégrité du système se manifeste par un flux d'informations illégal à un niveau de granularité défini : ce peut être le changement de la valeur d'une variable dans l'application, la lecture d'un fichier et l'envoi de son contenu par l'intermédiaire d'une socket, l'exécution d'un nouveau programme, etc. Certaines intrusions ne peuvent avoir d'effets qu'au niveau de l'application elle-même comme la modification d'une ou plusieurs variables et n'ont pas d'impact sur les flux d'informations au niveau du système d'exploitation. Cependant, de manière générale, les cibles intéressantes pour un attaquant peuvent être les fichiers et les processus du système d'exploitation et les moyens pour extraire de l'information du système, de manière distante, passent par des sockets. Dans ces cas, un flux d'informations illégal au niveau du système d'exploitation sera créé.

Le taux de couverture de l'approche n'est pas complet puisque nous ne considérons que les flux d'informations au niveau du système d'exploitation ce qui peut conduire à des faux négatifs. L'approche pourra également générer des faux positifs car les flux d'informations au niveau du système d'exploitation ne sont pas toujours semblables pour tous les COTS.

Dans la suite, nous appelons flux d'informations, un flux d'informations au niveau du système d'exploitation.

2.3.2 Graphes de flux d'informations

Nous appelons graphe de flux d'informations un ensemble de flux d'informations et d'objets impliqués dans ces flux qui ont été créés lors d'un appel au service surveillé. Un graphe de flux d'informations est un graphe orienté étiqueté c'est-à-dire que des étiquettes (une chaîne de caractères, un entier, etc.) sont associées aux nœuds et aux arcs.

Les objets du système d'exploitation (processus, fichiers, sockets, ...) sont les nœuds du graphe et les flux d'informations les arcs du graphe. Les étiquettes sont des informations relatives aux nœuds et aux arcs. Les étiquettes ont un rôle dans le calcul de la similarité comme nous le verrons dans la section 2.3.3. Dans notre prototype, les étiquettes possibles sont les suivantes :

- un type est associé aux nœuds : PROCESS, SOCKET, FILE, PIPE, MAPPING ;

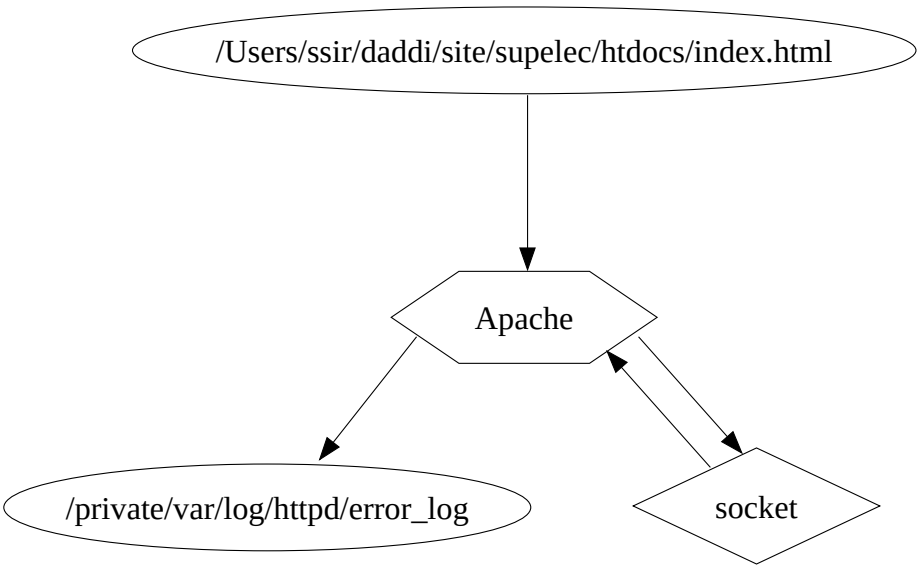


FIGURE 2.6 – Graphe de flux d'informations pour une requête HTTP donnée traitée par le serveur Apache sur Mac-OS X

– un type est également associé aux arcs : PROCESS_TO_FILE, FILE_TO-_PROCESS, PROCESS_TO_SOCKET, SOCKET_TO_PROCESS, PROCESS_TO_PIPE, PIPE_TO_PROCESS, PROCESS_TO_LOGFILE, EXECUTE.

D'autres données sont associées avec les nœuds et les arcs mais ne sont pas considérés comme des étiquettes dans le sens où elles ne sont pas utilisées dans le calcul de la similarité. Les nœuds du type FILE, c'est-à-dire les fichiers, sont associés à un nom, un descripteur de fichier, une date de création et de destruction du descripteur de fichier dans l'OS. Les nœuds de type PROCESS, c'est-à-dire les processus, sont associés avec leur nom, leur pid, le pid de leur parent et la date de création et de destruction du processus dans l'OS. Les arcs, c'est-à-dire les flux d'informations, sont associés avec les données transférées entre la source et la destination du flux et avec un temps d'accès.

En pratique, un processus o qui accède à un fichier f en lecture crée un flux d'informations avec deux nœuds o et f, liés par un arc étiqueté par un type et associé aux données lues par le processus o et le temps d'accès.

La figure 2.6 présente un exemple de graphes de flux d'informations. Ce graphe de flux d'informations représente le traitement d'une requête HTTP par le serveur web Apache. Les étiquettes et autres données associées ne sont pas toutes représentées pour éviter de surcharger la figure. Le type des objets est représenté par la forme des nœuds :

– Les sockets sont représentées par des losanges.

– Les processus sont représentés par des hexagones. Le nom du processus est inscrit dans l'hexagone.
– Les fichiers sont représentés par des ellipses. Le nom du fichier est inscrit dans l'ellipse.

Le graphe de la figure 2.6 montre que le processus Apache lit une socket, lit le fichier *index.html*, écrit dans le fichier de log *access.log* et écrit sur la socket des en-têtes HTTP puis le fichier *index.html* (la chronologie des flux d'informations est donnée par les temps d'accès associés aux flux ; ces temps d'accès ne sont pas représentés sur la figure).

2.3.3 Similarité entre deux graphes de flux d'informations

L'IDS est en charge de la comparaison des graphes de flux d'informations des différents COTS. Il est donc nécessaire d'avoir une méthode pour évaluer si deux graphes de flux d'informations sont similaires ou non. Pour cela, nous avons décidé d'utiliser le modèle développé par Champin et Solnon [CS03], qui permet de mesurer la similarité entre deux graphes étiquetés. Ces travaux peuvent ainsi s'appliquer aux graphes de flux d'informations que nous considérons. Dans ces travaux, ils donnent une définition numérique de la similarité entre deux graphes étiquetés et proposent deux algorithmes pour la calculer. Nous allons présenter leur approche et détailler l'algorithme que nous utilisons pour comparer des graphes de flux d'informations.

2.3.3.1 Définition formelle d'un graphe étiqueté

Un graphe étiqueté est un graphe orienté $G = (V, r_V, r_E)$ où :
– V est l'ensemble des nœuds ;
– $r_V \quad V \quad L$ est une relation entre les nœuds du graphe et les étiquettes des nœuds (L_V est l'ensemble des étiquettes des nœuds) ; (v, l) appartient à r_V si et seulement si v possède l'étiquette l ;
– $r_E \quad V \quad V \quad L_E$ est une relation entre les arcs et les étiquettes des arcs (L_E est l'ensemble des étiquettes des arcs) ; (u, v, l) appartient à r_E si et seulement si l'arc (u, v) possède l'étiquette l.

On note E l'ensemble des arcs ($E \quad V \quad V$). Le descripteur d'un graphe étiqueté $G = (V, r_V, r_E)$ est défini par $desc(G) = r_V \cup r_E$.

Dans notre cadre des graphes de flux d'informations, V contient un ensemble d'objets du système d'exploitation, E un ensemble de flux d'informations entre ces objets, et L_V et L_E sont les ensembles suivants :

$$L_V = \{PROCESS, SOCKET, FILE, PIPE, MAPPING\}$$

$$L_E = \left\{ \begin{array}{l} PROCESS\_TO\_FILE,\ FILE\_TO\_PROCESS, \\ PROCESS\_TO\_SOCKET,\ SOCKET\_TO\_PROCESS, \\ PROCESS\_TO\_PIPE,\ PIPE\_TO\_PROCESS, \\ PROCESS\_TO\_LOGFILE,\ EXECUTE \end{array} \right\}$$

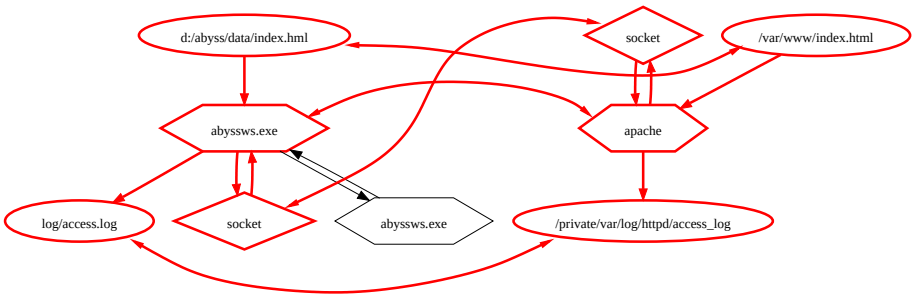


FIGURE 2.7 – Exemple de mapping entre deux graphes de flux d'informations : mapping entraînant une similarité forte

2.3.3.2 Similarité entre deux graphes en fonction d'un mapping

Considérons deux graphes étiquetés $G_1 = (V_1, r_{V_1}, r_{E_1})$ et $G_2 = (V_2, r_{V_2}, r_{E_2})$; nous cherchons à mesurer la similarité entre ces deux graphes.

Notion de mapping Pour cela, la notion de mapping est introduite : un mapping m est une relation $m \quad V_1 \quad V_2$. 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. Nous allons également définir la notion de mapping unitaire : m est un mapping unitaire s'il ne contient qu'un seul couple de nœuds, c'est-à-dire $Card(m) = 1$. Il faut remarquer que la notion de mapping se définit sur les nœuds et non sur les arcs. On dit qu'un arc $(v_{1,i}, v_{1,j})$ de G_1 est associé à un arc $(v_{2,k}, v_{2,l})$ de G_2 dans le mapping m si $v_{1,i}$ est associé à $v_{2,k}$ et $v_{1,j}$ à $v_{2,l}$ dans le mapping m.

De manière informelle, l'idée derrière la notion de mapping est d'associer des nœuds de G_1 et de G_2 qui ont le même rôle dans les deux graphes. Les figures 2.7 et 2.8 montrent deux exemples de mapping entre deux graphes de flux d'informations. Les graphes de flux d'informations sont les mêmes sur chaque figure et seuls les mappings sont différents. Le graphe de flux d'informations de gauche représente le traitement d'une requête HTTP par un serveur Abyss fonctionnant sous Windows 2000 Server ; celui de droite représente le traitement de la même requête HTTP par un serveur Apache fonctionnant sous MacOS-X. La figure 2.7 montre un mapping qui associe les nœuds ayant des « positions similaires » dans les deux graphes et les mêmes étiquettes. La figure 2.8 montre au contraire un mapping qui associe des nœuds des deux graphes ayant des étiquettes différentes et des « positions différentes » dans les graphes.

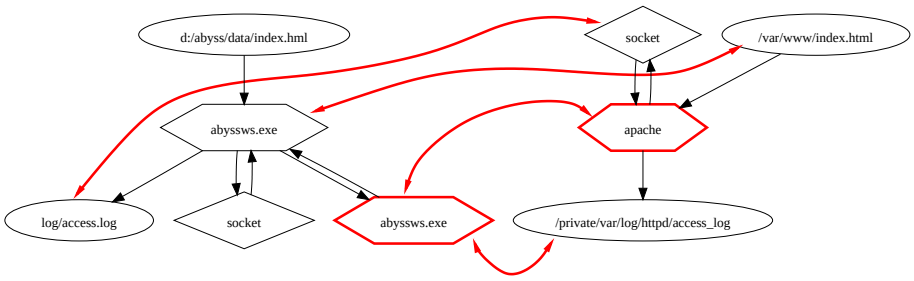


FIGURE 2.8 – Exemple de mapping entre deux graphes de flux d'informations : mapping entraînant une similarité faible

Similarité en fonction d'un mapping m La similarité entre G_1 et G_2 en fonction d'un mapping m est 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 (le cardinal est une fonction qui respecte ces critères par exemple) et $\sqcap_m$, appelé intersection en fonction du mapping m, est un opérateur ensembliste fonctionnant sur des descripteurs de graphe et est défini par la formule suivante :

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

Cet ensemble représente l'ensemble des éléments des descripteurs des deux graphes pour lesquels les étiquettes correspondent dans le mapping m. Cette formule est symétrique pour G_1 et G_2 : les lignes 1 et 3 correspondent à des éléments de G_1 alors que les lignes 2 et 4 correspondent à des éléments de G_2. Les lignes 1 et 2 correspondent aux nœuds alors que les lignes 3 et 4 correspondent aux arcs des graphes. Un couple (v,l) appartient à cet ensemble si et seulement si v est associé, dans le mapping m, à un nœud v' qui a la même étiquette l. La même relation existe pour les arcs : un triplet (v_i,v_j,l) appartient à cet ensemble si et seulement si l'arc (v_i,v_j) est associé, dans le mapping m, à un arc (v'_i,v'_j) qui possède la même étiquette l.

Le dénominateur de la formule de la similarité en fonction d'un mapping est constant si les graphes G_1 et G_2 sont fixés. Il permet de normaliser le calcul de la similarité qui est une valeur comprise entre 0 et 1. Étant donné les contraintes sur la fonction f, plus le mapping m associe des nœuds et des arcs qui ont les mêmes étiquettes plus la similarité est grande.

Dans notre modèle de graphes de flux d'informations, les seules étiquettes que nous considérons pour le calcul de la similarité sont les types décrits précédemment ; les nœuds et les étiquettes ont une seule étiquette. Par abus de langage, on peut dire alors qu'un nœud ou un arc appartient à l'ensemble $desc(G_1) \sqcap_m desc(G_2)$. Sur les figures 2.7 et 2.8, les nœuds et les arcs qui appartiennent à cet ensemble sont marqués en gris et d'un trait plus épais. Sur la figure 2.7, le mapping est intéressant car de nombreux nœuds et arcs appartiennent à l'ensemble déterminant le numérateur. Ce n'est pas le cas sur la figure 2.8 où le mapping associe des nœuds qui n'ont pas les mêmes étiquettes. La similarité en fonction du mapping représenté sur la figure 2.7 est donc supérieure ou égale à la similarité en fonction du mapping représenté sur la figure 2.8.

Notion de splits Cette définition de la similarité en fonction d'un mapping peut aboutir à des résultats surprenants et problématique : une similarité égale à 1 est éventuellement possible même si les deux graphes sont différents ; le mapping complet, c'est-à-dire qui associe tous les nœuds de G_1 à tous les nœuds de G_2 et réciproquement conduit à la similarité la plus élevée parmi tous les mappings. Pour résoudre ces deux problèmes, il est nécessaire d'introduire un dernier concept pour calculer la similarité entre deux graphes. Un nœud peut en effet être associé avec plusieurs autres nœuds dans un mapping. Ce concept est intéressant dans notre cas car il permet d'associer plusieurs processus sur un système d'exploitation donné avec un seul processus sur un autre système d'exploitation 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 :

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

Sur la figure 2.8, l'ensemble $splits(m)$ (m étant le mapping représenté sur la figure) contient un seul nœud : le nœud du graphe placé à gauche qui est marqué en gras. Ce nœud qui représente le processus père du serveur Abyss est en effet associé à deux nœuds du graphe de droite : le processus Apache et le fichier de log *access.log*.

La définition de la similarité en fonction d'un mapping m est alors modifiée pour prendre en compte la notion de *splits* :

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

où g est une fonction positive, croissante en fonction de l'inclusion. Le cardinal est une fonction qui vérifie ces propriétés ; la fonction constante égale à 0 vérifie également ces propriétés et permet de retrouver la première définition de la similarité en fonction d'un mapping. Ainsi deux graphes différents ne peuvent plus avoir une similarité de 1 et le mapping complet n'est plus forcément le mapping qui donne la similarité la plus élevée entre les deux graphes.

Si l'on choisit les fonctions f et g égales au cardinal, la similarité en fonction du mapping représenté sur la figure 2.7 est égale à $\frac{16}{19} \quad 0.842$; la similarité en fonction du mapping représenté sur la figure 2.8 est égale à $\frac{1}{19} \quad 0.053$.

Dans le cadre des graphes de flux d'informations, cette notion de *splits* introduit une nouvelle question : comment considérer les flux d'informations existant entre deux processus d'un graphe qui sont associés, tous les deux, à un même processus de l'autre graphe dans un mapping. Ce cas est visible sur la figure 2.9 où les deux flux d'informations entre les deux processus du serveur Abyss sont représentés en noir. Ces flux d'informations représentent des communications entre ces deux processus qui correspondraient éventuellement à des flux internes au processus Apache avec lequel sont associés les deux processus Abyss. Dans le modèle défini, ces flux d'informations ne sont associés à aucun flux alors qu'il est possible de les considérer comme associés à des flux internes au processus Apache qui ne sont pas visibles du point de vue du système d'exploitation. Il y a trois possibilités pour gérer ces flux : rester fidèle au modèle défini et les considérer comme non-associés, créer des flux du même type du processus Apache vers lui-même et les considérer donc comme associés, les considérer comme non-existant. Les deux dernières solutions modifient le caractère constant du dénominateur de la formule de similarité en fonction d'un mapping. Dans notre prototype, nous avons choisi la dernière solution qui est la solution médiane de manière générale.

Similarité entre deux graphes À 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 :

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

La similarité entre deux graphes est le maximum sur l'ensemble des mappings possibles des similarités en fonction d'un mapping. Trouver la similarité entre deux graphes, c'est trouver un mapping m qui maximise cette valeur.

Il est alors possible de définir la notion de meilleur mapping : un meilleur mapping m est un mapping maximal. La notion de meilleur mapping dépend des définitions des fonctions f et g. Cette notion est intéressante pour nous car c'est à partir d'un meilleur mapping que l'on cherchera à expliquer éventuellement les anomalies.

Si l'on choisit pour f et g le cardinal, un meilleur mapping entre les graphes des figures 2.7 ou 2.8 est représenté sur la figure 2.9. La similarité entre ces graphes est alors égale à $\frac{16}{17} \quad 0.941$.

Algorithmes de calcul de la similarité Le problème du calcul de la similarité entre deux graphes est plus général que le problème d'isomorphisme de graphes qui est lui-même un problème NP-complet. Champin et Solnon [CS03] ont proposé deux algorithmes pour résoudre ce problème : un algorithme de recherche complète avec des optimisations pour couper les branches de l'arbre

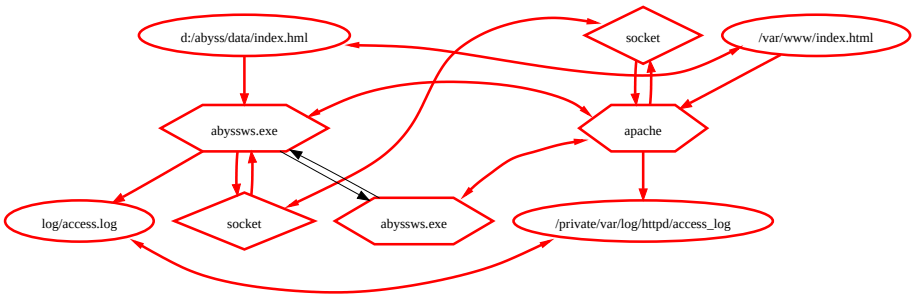


FIGURE 2.9 – Exemple d'un meilleur mapping entre deux graphes de flux d'informations

de recherche et un algorithme glouton. L'algorithme de recherche complète est en temps exponentiel en fonction du nombre de nœuds des graphes mais permet d'obtenir un meilleur mapping donnant la similarité entre les deux graphes. L'algorithme glouton a une complexité polynomiale en fonction du nombre de nœuds des graphes mais ne permet pas d'obtenir, de manière certaine, un meilleur mapping.

Nous avons choisi d'utiliser l'algorithme de recherche complète dans notre prototype. Nous sommes en effet plus intéressés par la précision du calcul de la similarité plutôt que par la rapidité de l'algorithme. L'implémentation de l'algorithme glouton proposé est cependant envisagée.

De plus, nous avons introduit certaines optimisations adaptées aux cas des graphes de flux d'informations :

- Cela n'a pas de sens d'associer un processus à un fichier, une socket ou un pipe ; ces mappings sont donc interdits.
- Les fichiers peuvent être utilisés comme moyen de communication entre processus. Cependant cette manière de faire n'est pas très performante et n'est donc utilisée que dans certains cas relativement précis. Nous considérons que les associations entre une socket ou un pipe et un fichier ne sont pas permises dans un mapping.
- Pour les serveurs réseau que nous considérons, nous pouvons envisager un certain nombre de points fixes : les sockets recevant les requêtes sont associées ensemble au début de l'algorithme, ainsi que les processus lisant ces sockets. Les fichiers de log sont également associés ensemble. Ces mappings sont établis au début de l'algorithme et les opérations de *backtracking* de l'algorithme de recherche complète ne sont pas autorisées à les enlever des différents mappings testés.

Grâce à ce modèle et ces algorithmes, il est possible d'obtenir une mesure de la similarité entre deux graphes de flux d'informations. L'IDS de notre approche de type boîte grise va donc utiliser ce modèle et l'algorithme de recherche

complète pour comparer les graphes de flux d'informations des différents COTS.

2.3.4 Détection d'intrusions par calcul des similarités

Ce modèle permet d'obtenir une valeur représentant la « ressemblance » entre deux graphes de flux d'informations. Plus la similarité est proche de 1, plus les graphes de flux d'informations sont semblables. Ces graphes de flux d'informations représentent le comportement du COTS en termes de flux d'informations au niveau du système d'exploitation.

À ce niveau de granularité, les flux d'informations pour des COTS fournissant des services réseau comme les services HTTP, FTP, POP3 sont semblables dans le cas de requêtes normales. Les graphes de flux d'informations doivent alors être sensiblement les mêmes sur les différents serveurs et donc la similarité doit être « élevée » dans le cas de requêtes normales.

2.3.4.1 Intrusions et graphes de flux d'informations

Nous considérons ici les intrusions qui ont un effet sur les flux d'informations au niveau du système d'exploitation. Les autres intrusions ne peuvent être détectées par notre modèle. Une intrusion se caractérise par des flux d'informations illégaux entre des processus et des fichiers, des sockets, des pipes, etc. Dans un graphe de flux d'informations, une intrusion sera responsable de la création ou de la modification de flux d'informations, d'objets actifs et/ou d'objets passifs.

Une intrusion contre la confidentialité sera caractérisée par la création de flux d'informations d'objets passifs vers des objets actifs et, si nécessaire, par la création de nouveaux objets. Par exemple, l'intrusion peut consister en l'ouverture d'un fichier, la lecture du contenu de ce fichier et l'envoi des données à l'attaquant.

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. Par exemple, l'intrusion peut consister en l'ouverture d'un fichier et l'écriture dans ce fichier d'un contenu provenant du réseau. Un autre exemple possible est la création d'un nouveau processus qui va ouvrir une socket pour télécharger un exécutable de type cheval de Troie.

Une intrusion contre la disponibilité (autre que les dénis de service) sera caractérisée par la création ou la disparition de flux d'informations ou d'objets actifs ou passifs. Par exemple, l'intrusion peut consister à faire défaillir le processus : il y a alors disparition de l'objet du processus et éventuellement de certains flux d'informations. Un autre exemple est la consommation excessive de ressources telles que le processeur ou la mémoire : une intrusion ayant cet objectif peut entraîner la disparition de certains flux d'informations comme l'écriture dans le fichier de log par exemple.

Une intrusion affecte le graphe de flux d'informations d'un serveur compromis. La valeur de la similarité entre les graphes de flux d'informations d'un serveur compromis et d'un serveur non-compromis peut donc diminuer en cas d'intrusion.

2.3.4.2 Notion de seuil de similarité

Une valeur élevée de la similarité signifie que les serveurs se sont comportés d'une manière identique en ce qui concerne les flux d'informations. 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 n'est pas possible de distinguer, à partir de la similarité entre les graphes de flux d'informations, une intrusion d'une différence de conception ou de spécification. Cette approche est donc susceptible de générer des faux positifs.

Plusieurs approches peuvent être utilisées pour décider quand lever une alerte ou non. Nous avons choisi d'utiliser une solution binaire en choisissant un seuil de similarité t en-dessous duquel une alerte est levée. t est strictement inférieur à 1 à cause des différences de conception : nous souhaitons qu'au-dessus de ce seuil, il y ait une forte probabilité que la différence détectée soit due à une différence de conception. En-dessous de ce seuil, la différence peut s'expliquer soit par une intrusion soit par une différence de conception ou de spécification importante.

Une approche probabiliste peut être envisagée en calculant à partir de la similarité une probabilité d'intrusion. La fonction de calcul de probabilité d'intrusion en fonction de la similarité doit vérifier deux contraintes : $f(0) = 1$ et $f(1) = 0$. Il est possible d'imposer des contraintes supplémentaires : $f(t) = \frac{1}{2}$ ou les dérivées n$^{\text{es}}$ en 0 nulles. La fonction $f(x) = \frac{1}{2(t^2)}\ 2x + \frac{2t^2}{2(t^2)}\ x + 1$ vérifie les trois contraintes : $f(0) = 1$, $f(t) = \frac{1}{2}$ et $f(1) = 0$; c'est tout simplement le polynôme d'interpolation de Lagrange passant par ces trois points.

Pour déterminer t, nous avons décidé de procéder par apprentissage dans un contexte normal d'utilisation. Se superposent aux problèmes d'apprentissage classique (absence d'attaques dans la base d'apprentissage notamment) un problème lié aux différences de conception ou de spécification. Il est préférable que la base d'apprentissage ne contiennent pas de requêtes entraînant des différences de spécification ou de conception trop importantes entre les serveurs COTS. De même qu'il est difficile de vérifier que la base d'apprentissage est exempte d'intrusions, il est difficile de vérifier qu'elle ne contient pas de requêtes entraînant des similarités « basses » dues à des différences de conception ou de spécification. Il n'est ainsi pas possible de choisir t comme le minimum des similarités calculées lors de la phase d'apprentissage. Nous avons alors décidé de déterminer t comme une valeur qui permet qu'un pourcentage élevé (supérieur à 99% par exemple) des requêtes de la base d'apprentissage ne créent pas d'alertes.

Il faut noter que le seuil de similarité t dépend des couples de COTS utilisés : il doit être donc déterminé pour chaque couple de services diversifiés. Il est intéressant de le déterminer non seulement en fonction des serveurs COTS mais également en fonction du type de requêtes. La similarité dépend en effet du cardinal des descripteurs des graphes considérés et dépend de l'application surveillée mais aussi des requêtes. 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'informations non associés auront moins d'influence sur le calcul de la similarité que si les

graphes sont petits (c'est-à-dire ont peu de nœuds et d'arcs). Il est cependant difficile de classer tous les types de requêtes : dans le cas d'un serveur web, la réponse et le traitement de la requête vont dépendre d'un éventuel contexte (authentification par exemple), de la ressource accédée et des paramètres de la requête. Il est plus simple de définir différents seuils de similarité en fonction du cardinal des descripteurs : $Card(desc(G_1) \cup desc(G_2))$ en choisissant par exemple des intervalles pour lesquels chaque seuil est valable.

2.3.4.3 Algorithmes de détection

En déterminant le seuil t, nous avons donc une méthode pour décider quand lever une alerte lorsque l'IDS compare les graphes de flux d'informations de deux serveurs. Dans le contexte d'une architecture à $N > 2$ serveurs COTS S_i (i dans $\{1, N\}$), il faut déterminer un seuil $t_{i,j}$ pour chaque couple de serveurs comme expliqué précédemment. $t_{i,j}$ est le seuil choisi pour les serveurs S_i et S_j. Notons $G_{i,e}$ le graphe de flux d'informations du serveur S_i pour la requête e, $s_{i,j,e}$ la similarité entre le graphe $G_{i,e}$ et $G_{j,e}$. Comme la similarité est symétrique, nous avons $s_{i,j,e} = s_{j,i,e}$ pour tout (i, j) dans $\{1, N\}^2$ et pour tout e dans l'espace des requêtes. Appelons la relation « être semblable à », la relation entre deux graphes de flux d'informations notée et définie par :

$$G_{i,e} \quad G_{j,e} \Leftrightarrow s_{i,j} \quad t_{i,j}$$

La relation « être semblable à » est symétrique mais n'est pas a priori une relation transitive : rien ne prouve en effet que si $G_{i,e}$ est semblable à $G_{j,e}$ et $G_{j,e}$ est semblable à $G_{k,e}$, alors $G_{i,e}$ est semblable à $G_{k,e}$. Il n'est alors pas possible dans ce cas d'établir des classes d'équivalence entre les graphes de flux d'informations.

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 $C_N^2 = \frac{N!}{(N \quad 2)! \quad 2!}$ similarités entre graphes de flux d'informations.

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 lever une alerte dans le cas de N serveurs :

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

c'est-à-dire qu'une alerte est émise dès qu'une similarité $s_{i,j,e}$ est faible (en-dessous de $t_{i,j,e}$). Aucune alerte n'est émise seulement lorsque toutes les similarités calculées sont supérieures à leur seuil respectif. Ces règles correspondent aux règles formulées dans le cas général (voir 2.1.2.2). Ces règles sont strictes et peuvent éventuellement générer des faux positifs dans des cas limites : par exemple, dans le cas où tous les graphes sont semblables deux à deux sauf pour un couple de graphes où la similarité calculée est juste en-dessous du seuil de similarité.

D'autres règles peuvent être suivies pour limiter ces cas. Notons $m_{i,e}$ la moyenne des similarités qui concernent le serveur S_i : $m_{i,e} = \frac{\sum_{j \neq i} s_{i,j,e}}{N}$ et notons t_i la moyenne des seuils qui concernent le serveur S_i : $t_i = \frac{\sum_{j \neq i} t_{i,j}}{N}$. Les règles suivantes peuvent alors être utilisées pour déterminer si une alerte doit être levée ou non :

$$\exists i \in \{1, N\}, m_{i,e} < t_i \Rightarrow Alerte$$
$$\forall i \in \{1, N\}, m_{i,e} \not< t_i \Rightarrow Pas\ d'alerte$$

Ces règles sont à rapprocher des règles émises pour la localisation du serveur compromis dans la section 2.1.2.2.

2.3.4.4 Localisation du serveur compromis

Le fait que la relation « être semblable à » n'est pas une relation transitive pose certaines difficultés : il n'est pas possible dans tous les cas d'établir des classes d'équivalence entre les graphes. Dans les cas où il est possible de trouver une classe d'équivalence entre des graphes semblables qui forme une majorité (son cardinal est supérieur à $\frac{N}{2}$), la règle classique peut être utilisée : tous les serveurs non présents dans cette classe d'équivalence sont considérés comme compromis. La figure 2.10 montre un cas où il n'est pas possible de trouver de classe d'équivalence formant une majorité : les graphes G_1 et G_2 sont semblables, le graphe G_3 est semblable à G_1 et G_2 mais n'est pas semblable à G_4 ; de la même manière, G_4 est semblable à G_1 et G_2 mais non à G_3 ; le graphe G_5 n'est semblable à aucun des autres graphes.

Dans le cas présenté sur la figure 2.10, la règle générale indique que la localisation n'est pas possible : tous les serveurs doivent être considérés comme compromis. Ceci entraîne une indisponibilité de l'architecture. Pourtant il est possible de trouver deux majorités : G_1, G_2 et G_3 peuvent constituer une majorité tout comme G_1, G_2, G_4. Des règles de localisation moins strictes peuvent alors être choisies. Il est possible de choisir aléatoirement entre G_3 et G_4 pour obtenir une majorité et considérer l'autre serveur compromis. Il est possible d'établir d'autres règles de localisation :

$$\exists i \in \{1, N\}, Card(G^{i,e}) > \frac{N}{2} \Rightarrow S_i \text{ est compromis}$$

où $G^{l,e}$ est l'ensemble $\{G_{k,e} | k \in \{1, N\} \setminus \{l\}, s_{l,k,e} \in I_1^{l,k}\}$.

Avec cette règle, sur l'exemple de la figure 2.10, seul le serveur S_5 dont le graphe est G_5 est considéré comme compromis.

En modifiant seulement, le signe de comparaison, il est possible d'obtenir une localisation plus stricte :

$$\exists i \in \{1, N\}, Card(G^{i,e}) \not> \frac{N}{2} \Rightarrow S_i \text{ est compromis}$$

Avec cette règle, les serveurs G_3, G_4 et G_5 sont considérés comme compromis dans l'exemple de la figure 2.10.

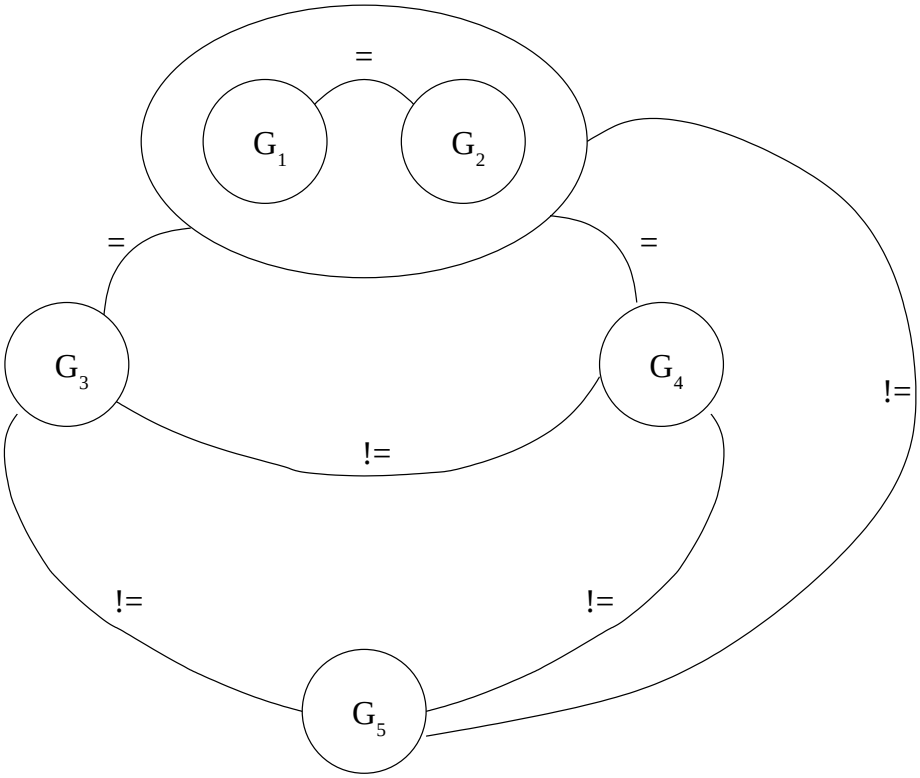


FIGURE 2.10 – Problème lié à la non-transitivité de la relation « être semblable à » ; chaque nœud représente un graphe, = signifie que de graphes sont semblables et ! = signifie que les graphes ne sont pas semblables.

Ces deux règles signifient que les serveurs considérés comme compromis sont ceux dont le graphe de flux d'informations n'est pas semblable à plus de la moitié des autres graphes.

2.3.4.5 Cas de trois serveurs

Le tableau 2.5 résume, dans le cas de trois serveurs S_1, S_2 et S_3, dans quels cas une alerte doit être levée (dans le cas de l'utilisation des premières règles de détection) et s'il est possible de localiser le serveur compromis en fonction des similarités calculées. Dans le cas de trois serveurs, les trois règles de localisation proposées sont identiques.

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

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

TABLE 2.5 – 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 S_i est considéré comme compromis

du calcul de similarité un serveur dont le comportement est différent des deux autres.

Nous avons présenté la méthode de comparaison de notre IDS de type boîte grise. Cette comparaison est fondée sur le calcul de la similarité entre les graphes de flux d'informations des serveurs COTS. Cette méthode permet de prendre en compte des intrusions qui seraient invisibles à des approches de type boîte noire comme celle que nous avons présentée auparavant. Un autre intérêt de cette approche est d'apporter un premier diagnostic de l'anomalie détectée.

2.3.5 Aide au diagnostic

Les graphes de flux d'informations représentent schématiquement les flux d'informations se déroulant au niveau du système d'exploitation. L'algorithme de recherche complète permettant le calcul de la similarité entre deux graphes de flux d'informations donne également un meilleur mapping entre ces deux graphes. Nous allons utiliser ce meilleur mapping pour aider l'opérateur à établir un diagnostic d'une alerte. L'algorithme de recherche complète est capable de nous donner la liste des meilleurs mappings entre deux graphes : nous avons décidé de considérer le premier mapping trouvé par l'algorithme dont le cardinal est minimal comme le meilleur mapping.

Cette aide au diagnostic repose sur les objets actifs (processus), les objets passifs (sockets, fichiers, pipes, ...) et les flux d'informations non-associés dans le meilleur mapping.

En cas de vrai positif, les objets et flux d'informations non associés dans le meilleur mapping sont les effets visibles au niveau du système d'exploitation de l'exploitation d'une vulnérabilité. L'analyse de ces objets et flux d'informations permet d'obtenir des informations sur l'intrusion : processus créés, fichiers lus ou écrits, sockets créées, etc. S'il n'est pas possible de retrouver directement la vulnérabilité exploitée, ces informations peuvent y conduire. Dans le cas d'un zero-day, cela peut constituer un bon point de départ pour découvrir la vulnérabilité inconnue.

En cas de faux positif, ces objets et flux d'informations seront l'explication de la différence de conception ou de spécification qui a eu pour conséquence la

levée d'une alerte.

Nous proposons de montrer à l'administrateur de sécurité du système les graphes des différents serveurs pour la requête suspecte en mettant en valeur les objets et les flux non associés dans le meilleur mapping. Nous reprenons ici, en partie, l'idée présentée par King et Chen [KC03, KC05] et également repris plus tard dans Sebek [Pro03, BV05]. L'idée proposée par King et Chen est de présenter un graphe de flux d'informations à l'administrateur de sécurité en cas d'intrusion car ce graphe, s'il n'est pas trop grand, peut être facilement interprété par l'administrateur. Balas et Viecco [BV05] reprennent ce principe en l'intégrant dans Sebek, un outil pour Honeypot. Dans notre approche, nous présentons les graphes de flux d'informations de chaque serveur en mettant en valeur (en coloriant par exemple) les objets ou les flux d'informations non associés. Des exemples d'aide au diagnostic que peut apporter cette méthode sont présentées dans la section 4.5.

Nous avons présenté le modèle général de détection d'intrusions et ses propriétés puis les spécificités de nos deux approches de détection par diversification de COTS, tout en mettant en évidence les avantages et limitations de l'approche globale et des deux approches. Dans la section suivante, nous comparons plus en détail les deux approches et présentons comment il est possible de corréler leurs alertes.

2.4 Comparaison des deux approches proposées

Les deux approches que nous proposons peuvent être utilisées en collaboration pour améliorer la couverture de détection de l'architecture. C'est ce que nous présentons tout d'abord. Nous discutons ensuite des différences entre les deux approches.

2.4.1 Corrélation entre les approches par comparaison des sorties et calcul de similarité

Même si beaucoup de travaux [WFMB03, RM08] ont porté sur la collaboration ou la composition d'IDS, notre cas est relativement simpliste puisque nous ne considérons que deux IDS. Nous utilisons donc une simple approche ensembliste. Le tableau 2.6 résume la décision prise par la combinaison en cas de désaccord. 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. Cette solution est logique dans le sens où chacune des deux approches a une couverture de détection en partie différente alors que les faux positifs vont être principalement générés, dans le cadre des deux approches, par les différences de spécification ou de conception. Ces différences risquent souvent d'avoir à la fois une influence sur les flux d'informations et sur les sorties réseau.

IDS boîte noire \ IDS boîte grise	alerte	pas d'alerte
alerte	alerte	alerte
pas d'alerte	alerte	pas d'alerte

TABLE 2.6 – Corrélation entre l'approche boîte noire et l'approche boîte grise

2.4.2 Discussion

Les deux approches que nous avons proposées présentent des différences que nous résumons ici.

La méthode de masquage des différences de spécification est naturelle dans le cas de l'approche de type boîte noire. Elle peut également s'appliquer dans le cas de l'approche de type boîte grise. Cela est cependant plus complexe. Les observations sont, en effet, dans ce cas, des graphes de flux d'informations. La méthode que nous proposons pour les comparer est coûteuse puisque l'algorithme utilisé est exponentiel en temps. Si la méthode de masquage choisie oblige à calculer de nouvelles similarités, cela va entraîner de fortes dégradations de performance. En outre, le modèle utilisé permet de masquer des différences de conception ou de spécification mineures grâce à la notion de seuil de similarité.

Le choix du moment des observations est naturel pour l'approche boîte noire : à chaque fois que l'interface standardisée est utilisée. Ce n'est pas toujours le cas pour l'approche de type boîte grise. Pour le protocole HTTP, les requêtes/réponses sont bien délimitées et il est naturel de comparer les graphes de flux d'informations après l'envoi des réponses au proxy. Les cas des protocoles FTP et POP3 sont plus complexes. Les sessions peuvent être très longues et peuvent comporter beaucoup de requêtes/réponses. Comparer les graphes de flux d'informations pour chaque couple requête/réponse de la session peut, d'une part, être pénalisant pour les performances et, d'autre part, non significatif car les graphes peuvent être limités à des lectures et écritures sur la socket. Si l'on décide de ne comparer les graphes de flux d'informations qu'à la fin de la session ou tous les X couples de requêtes/réponses, il reste à déterminer quelle réponse envoyer pour chaque requête, ce qui pose problème si seul l'IDS boîte grise est utilisé. Une solution, dans ce cas, est de choisir un leader dont on enverra les réponses au client.

On peut remarquer que l'approche de type boîte noire est presque « contenue » dans l'approche de type boîte grise. En effet, les sorties réseau comparées par la première approche sont les flux d'informations en écriture d'un processus vers la socket ou les sockets. Comme on ne considère dans le modèle que le type des flux d'informations, cette inclusion n'est pas valide mais il est envisageable de définir les étiquettes sur les flux d'informations en écriture sur les sockets à partir de l'approche boîte noire : les étiquettes des flux d'informations en écri-

ture sur les sockets seraient les mêmes pour les sorties réseau considérées comme équivalentes par l'IDS boîte noire. La coopération entre les deux IDS seraient ainsi renforcés et la décision de l'IDS boîte noire intégrée à l'IDS boîte grise.

2.5 Résumé

Nous avons proposé une architecture de détection d'intrusions fondée sur la diversification de COTS, qui reste très similaire à celle d'autres projets. Nous en avons étudié différents aspects pour mettre en évidence les avantages et les limites. Nous avons formalisé les propriétés de détection d'intrusions, de localisation des serveurs compromis et de tolérance aux intrusions dans le cadre de la diversification de COTS en distinguant deux cas : observations parfaites ou non. Nous avons ensuite présenté les deux approches de détection d'intrusions, que nous avons proposées, fondées sur cette architecture.

Notre approche de type boîte noire se distingue de l'existant par un algorithme de détection précis et une méthode de masquage des différences de conception ou de spécification différente, qui permet de limiter les faux positifs même dans des cas où le comportement des COTS est très différent ou mal défini.

Notre approche de type boîte grise repose sur le calcul de similarité entre graphes de flux d'informations au niveau du système d'exploitation. Nous avons présenté le modèle et l'algorithme de calcul de la similarité. Nous avons discuté les choix faits lors de l'utilisation de ce modèle dans le cadre de la diversification de COTS pour la détection et la tolérance aux intrusions. En plus de ces capacités de détection, il est possible d'utiliser cette méthode pour aider l'administrateur à diagnostiquer les anomalies détectées.

Chapitre 3

Application aux serveurs web

« Profanity is the one language all programmers know best. »

Nous avons appliqué nos deux approches de détection au cadre des serveurs web. Comme expliqué dans la section 1.1.2, de nombreux IDS « génériques » ont été testés sur des serveurs web et de nombreux IDS spécifiques aux serveurs web ont été développés. En effet, le protocole HTTP est l'un des protocoles fondamentaux de l'Internet et un protocole universel qui est utilisé pour de nombreuses applications différentes : partage de fichiers, paiement électronique, etc. Pour toutes sortes d'applications, la sécurité des serveurs web est critique. Les serveurs web sont ainsi des cibles intéressantes pour les pirates, par exemple dans des buts de chantage ou comme moyen d'infection des clients web.

La sécurité des serveurs web et des applications fonctionnant sur ces serveurs est critique tant pour l'entité représentée par le serveur que pour les clients visitant ces serveurs.

Dans ce chapitre, nous présentons brièvement le protocole HTTP puis les implémentations de nos deux approches dans le cadre des serveur web. Notre méthode de détection d'intrusions repose sur la diversification, nous proposons finalement une méthode permettant de diversifier automatiquement les applications fonctionnant sur les serveurs HTTP pour détecter et tolérer des intrusions par injection de code.

Il est nécessaire de définir ce que nous entendons par serveur web statique et serveur web dynamique. Un serveur web statique est un serveur dont le contenu change peu et dont les pages sont codées « en dur » : les données envoyées au client ne sont pas créées dynamiquement lorsque le client envoie une requête. Un serveur web dynamique est un serveur dont le contenu change plus régulièrement. Les données envoyées au client sont créées dynamiquement à chaque requête. Les données sont généralement stockées dans une base de données et le serveur utilise des langages de scripts tels que php ou perl, par exemple, pour créer les pages dynamiquement.

```
GET / HTTP/1.0
```

FIGURE 3.1 – Requête HTTP valide simple

3.1 Description du protocole HTTP

Le protocole HTTP est un protocole de communication au niveau applicatif. Il a été conçu dans une optique de légèreté et de rapidité pour les systèmes d'information distribués et hypermédias. Il en existe trois versions : HTTP/0.9, HTTP/1.0 et HTTP/1.1 [FGM+99]. Le protocole est déconnecté et fondé sur le paradigme requête/réponse.

3.1.1 Format des requêtes HTTP

Un client établit une connexion avec un serveur et envoie une requête au serveur se composant d'une méthode, d'une URI (Uniform Ressource Identifier) indiquant l'entité concernée, de la version du protocole, suivi par un message de type MIME qui contient des éléments modifiant la requête, des informations sur le client et un corps de message si cela est nécessaire. De manière formelle, exprimé dans une grammaire BNF (forme de Backus-Naur) où l'on ne définit pas tous les éléments :

```
Request = Request-Line
          *(( general-header
          | request-header
          | entity-header) CRLF)
          CRLF
          [ message-body ]

Request-Line = Method SP Request-URI SP HTTP-Version CRLF
```

`CRLF` représente les deux caractères retour chariot (carriage return) et line feed (nouvelle ligne) du code ASCII. `SP` représente le caractère espace. Deux exemples de requêtes sont présentés sur les figures 3.1 et 3.2 :

Le premier exemple (figure 3.1) montre une requête valide extrêmement simple. La requête ne contient qu'une seule ligne, la ligne de requête.

Le second exemple (figure 3.2) montre une requête bien plus complexe contenant, en plus de la ligne de requête, des nombreux en-têtes et contenant un corps de message.

Le fait que les requêtes puissent avoir des formes très distinctes, comme le montrent les deux exemples, participe au caractère universel du protocole HTTP, qui fait que ce protocole est utilisé de multiples manières. De nombreux en-têtes sont définis et certains serveurs n'implémentent pas ou incorrectement certains en-têtes voire même certaines méthodes HTTP (qui sont, elles, en nombre plus restreint). La diversité importante des requêtes peut conduire

```
POST /bibtex/process.php HTTP/1.1
Connection: close
Accept: text/xml,application/xml,application/xhtml+xml,text/html;
                      q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Accept-Encoding: gzip,deflate
Accept-Language: fr,fr-fr;q=0.8,en-us;q=0.5,en;q=0.3
Host: 192.168.0.47
User-Agent: Mozilla/5.0 (X11; U; Linux i686; fr; rv:1.8.0.4)
      Gecko/20060406 Firefox/1.5.0.4 (Debian-1.5.dfsg+1.5.0.4-1)
Content-Length: 558
Content-Type: application/x-www-form-urlencoded
Cookie: PHPSESSID=6e99d382d8d9a9cd9777570d9d848f4f

bibtex=%40inproceedings%7BBLJJ05%2C%0D%0A++++author+%3D+%7B
David+A.+Bryan+and+Bruce+B.+Lowekamp+and+Cullen+Jennings%7D
%2C%0D%0A++++title+%3D+%7B%7BSOSIMPLE%7D%3A+A+Serverless%2C
+Standards-based%2C+%7BP2P+SIP%7D+Communication+System%7D%2C
%0D%0A++++booktitle+%3D+%7BProceedings+of+the+2005+Internati
onal+Workshop+on+Advanced+Architectures+and+Algorithms+for+
Internet+Delivery+and+Applications+%28AAA-IDEA+2005%29%7D%2C
%0D%0A++++year+%3D+%7B2005%7D%2C%0D%0A++++keywords+%3D+%7B
P2P%2C+SIP%7D%2C%0D%0A++++confidential+%3D+%7Bn%7D%0D%0A
print+%22echo%22%3B%0D%0A%7
```

FIGURE 3.2 – Requête HTTP valide plus complexe

tout naturellement à des différences de spécification entre les COTS que nous allons utiliser dans notre architecture de détection. Il est nécessaire d'introduire un mécanisme de normalisation des requêtes au niveau du proxy, comme indiqué au 2.1.4, pour essayer de limiter les requêtes à l'intersection des différentes spécifications des COTS.

3.1.2 Format des réponses HTTP

Le serveur répond à une requête par une ligne de statut se composant de la version du protocole et d'un code de succès ou d'erreur, suivi également par un message de type MIME contenant des informations sur le serveur, des méta-informations sur l'entité et un contenu si cela est nécessaire.

Une réponse peut être de deux types : soit simple soit complète. Les réponses simples ne contiennent que du contenu et doivent se limiter aux requêtes des clients ne supportant que la version 0.9 du protocole. Les réponses complexes contiennent une ligne de statut, des en-têtes généraux, des en-têtes liés à la réponse, des en-têtes liés à l'entité et le contenu de l'entité.

```
HTTP/1.1 200 OK
Date: Thu, 18 Sep 2008 13:22:00 GMT
Server: Apache/1.3.37 (Unix)
X-Powered-By: PHP/4.4.4
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html

22a
<html><header><title> BibTeX Database Manager </title>
<link href="css/style.css" rel="Stylesheet" type="text/css">
</header><body class="texte"><center><img src="images/banner2.gif"
width="700" height="50" /><table><tr><FORM NAME="usersAdd"
ACTION="index.php" METHOD="POST"><td>User</td><td><input type="text"
name="user" size="8"></td></tr><tr><td>Password</td><td>
<input type="password" name="pwd" size="8"></td></tr><tr>
<td COLSPAN=2 align="center"><input type="submit" name="login"
value="Login"></td></tr></form></tr></table></center></body></html>
0
```

FIGURE 3.3 – Réponse HTTP d'un serveur Apache

Une réponse complexe se définit de manière plus formelle à l'aide d'une grammaire BNF (où l'on ne définit pas tous les éléments) :

```
Response = Status-Line
           *(( general-header |
               response-header |
               entity-header ) CRLF)
           CRLF
           [ message-body ]

Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

La ligne de statut est la plus importante car elle renvoie notamment le code de réponse et donc comporte une grande partie du sens de la réponse. Ces codes sont classés comme suit :

- 1XX : codes d'information ;
- 2XX : codes indiquant le succès de la requête ;
- 3XX : codes indiquant la nécessité d'entreprendre d'autres actions pour compléter la requête ;
- 4XX : codes indiquant une erreur du client ;
- 5XX : codes indiquant une erreur du serveur.

Une liste de codes est définie dans la spécification ; il est également possible

```
HTTP/1.1 200 OK
X-Powered-By: PHP/4.4.6
Content-type: text/html
Connection: Close
Date: Thu, 18 Sep 2008 13:21:59 GMT
Server: Abyss/2.0.6-X1-Win32 AbyssLib/2.0.6

<html><header><title> BibTeX Database Manager </title>
<link href="css/style.css" rel="Stylesheet" type="text/css">
</header><body class="texte"><center><img src="images/banner2.gif"
width="700" height="50" /><table><tr><FORM NAME="usersAdd"
ACTION="index.php" METHOD="POST"><td>User</td><td><input type="text"
name="user" size="8"></td></tr><tr><td>Password</td><td>
<input type="password" name="pwd" size="8"></td></tr><tr>
<td COLSPAN=2 align="center"><input type="submit" name="login"
value="Login"></td></tr></form></tr></table></center></body></html>
```

FIGURE 3.4 – Réponse HTTP d'un serveur Abyss

d'étendre cette liste en ajoutant de nouveaux codes et leur sémantique.

Les en-têtes généraux s'appliquent spécifiquement au message HTTP transmis et contiennent des informations sur l'état de la connexion et sur le contrôle des mécanismes de cache par exemple.

Les en-têtes liés à la réponse contiennent des informations additionnelles qui n'ont pas pu prendre place sur la ligne de statut comme le nom du serveur répondant à la requête, la méthode d'authentification utilisée, la localisation exacte de la ressource demandée dans la requête...

Les en-têtes concernant l'entité comportent des informations sur le contenu de l'entité. Ils peuvent comporter notamment la taille du contenu, le type du contenu et l'encodage de celui-ci si cela est nécessaire, la dernière modification de l'entité sur le serveur, la langue du contenu (si celui-ci est textuel notamment), l'encodage utilisé pour le transport.

La figure 3.3 présente un exemple de réponse HTTP. Cette réponse contient la ligne de statut qui indique que la requête a été traitée avec succès par le serveur, des en-têtes qui indiquent la date, la version du serveur, par exemple, et un contenu qui est ici du HTML qui sera interprété par le navigateur. La figure 3.4 présente une autre réponse HTTP à la même requête. On peut remarquer que les deux réponses sont valides mais différentes. L'approche boîte noire que nous avons proposée est chargée de comparer ces réponses. C'est ce que nous détaillons dans la section suivante.

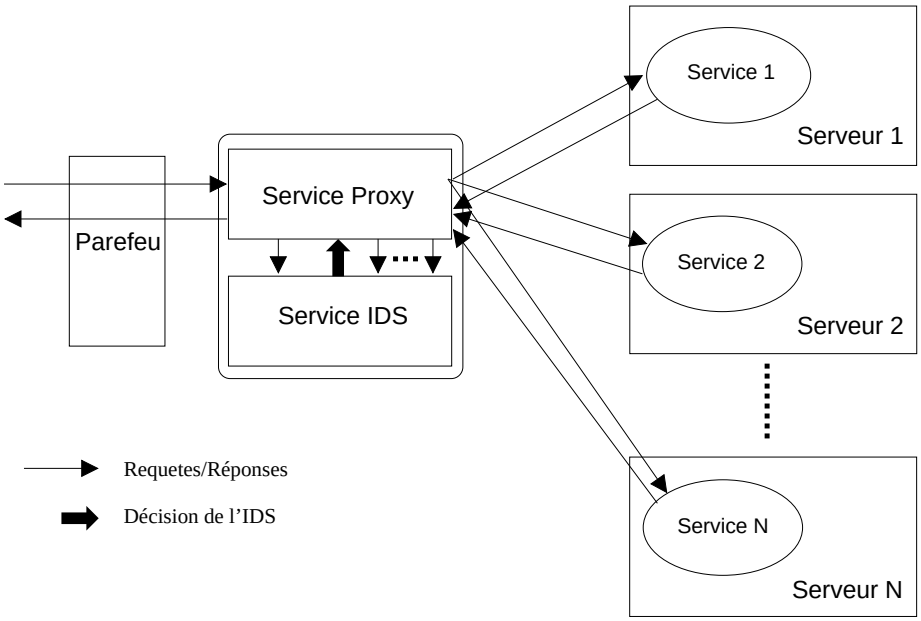


FIGURE 3.5 – Architecture de l'approche boîte noire

3.2 Approche boîte noire : description de l'implémentation

L'algorithme de comparaison des réponses est spécifique à chaque service ; un algorithme doit donc être développé pour chaque type de service. Nous exposons ici d'abord l'architecture implémentée, dans le cadre de notre approche boîte noire, pour les serveurs web puis l'algorithme de comparaison en lui-même et enfin l'implémentation du masquage des différences de spécification.

3.2.1 Architecture

Dans le cas de l'approche boîte noire, les observations sur le comportement des serveurs web sont leurs réponses HTTP ; il n'est pas nécessaire d'envoyer deux fois ces réponses. Le proxy les transmet donc à l'IDS (voir figure 3.5). De plus, le proxy transmet la requête à l'IDS ce qui est nécessaire pour le masquage des différences de spécification comme expliqué en 2.1.4. Après avoir comparé les réponses, l'IDS donne sa décision au proxy et lève éventuellement une alerte.

3.2.2 Algorithme de comparaison

Comme nous l'avons vu dans la section précédente, une réponse HTTP est composée de deux parties distinctes : les en-têtes HTTP et le corps. Le corps

contient les données demandées par le client si la requête est acceptée. Les en-têtes contiennent des informations sur la réponse : état de la réponse (refus ou acceptation de la requête), le nom du serveur, la date, le nombre d'octets transmis, etc. Certains en-têtes sont obligatoires, d'autres seulement facultatifs.

Une comparaison brute des réponses est impossible car les serveurs ne renvoient pas les mêmes en-têtes pour une même requête ni même exactement le même corps suivant l'encodage de transmission choisi (en-tête Transfer-Encoding). En effet, les en-têtes « Date » et « Server » risquent d'être différents pour chaque serveur. Cela se vérifie sur les exemples figures 3.3 et 3.4. Sur ces figures, on peut remarquer que le serveur Apache a choisi d'utiliser un encodage de transmission de type chunked alors que le serveur Abyss marque la fin de la réponse en fermant la socket.

Nous avons alors choisi de comparer les codes de statut de la réponse et les en-têtes : *Content-Length*, *Content-Type*, *Last-Modified*, *Location*. Il faut également vérifier la présence de l'en-tête ou l'absence de l'en-tête *Set-Cookie* dans la réponse de tous les serveurs. Nous avons également choisi de comparer le corps de la réponse. Il faut distinguer deux cas : le cas d'un serveur servant un contenu statique et celui d'un serveur servant un contenu dynamique.

Contenu statique Lorsque le contenu des serveurs est statique, il est possible de comparer de manière binaire le corps des réponses, en gérant toutefois les différents types d'encodage de transmission. Même dans le cas où le serveur web est statique, certains contenus peuvent être générés dynamiquement comme les pages d'erreur, la page d'erreur 404 par exemple, ou les pages listant le contenu de répertoire. Dans ces cas, le serveur web génère dynamiquement le code HTML de la réponse. Celui est généralement différent d'un serveur à l'autre et empêche une comparaison binaire. Certains serveurs donnent la possibilité de définir des pages d'erreurs pour chaque type d'erreur ; il est ainsi possible de comparer de manière binaire le corps des réponses même en cas d'erreur. Ce n'est cependant pas le cas de tous les serveurs. De la même manière, certains serveurs peuvent être configurés pour interdire aux serveurs de lister le contenu d'un répertoire automatiquement ; les serveurs répondent alors par un message d'erreur aux requêtes souhaitant le contenu d'un répertoire.

Contenu dynamique Dans le cas de serveurs web dynamiques, la comparaison des corps de réponse est encore moins évidente que dans le précédent cas. Une comparaison binaire peut poser certains problèmes lorsque des éléments spécifiques des serveurs tels que la date ou le nom du serveur sont présents dans le corps. C'est le cas par exemple pour des forums où la date des messages est présente dans le corps du message envoyé au client. Si aucun élément spécifique n'est contenu dans les réponses, la comparaison binaire est possible.

Pour résoudre le problème des éléments spécifiques, plusieurs solutions peuvent être envisagées :

- La première solution est de modifier l'application et ainsi éviter qu'elle n'envoie des données spécifiques des serveurs. Ceci n'est pas toujours pos-

sible si le code source de l'application n'est pas disponible ou si retirer ces informations empêche l'application de remplir correctement son service. Ceci permet de faire la comparaison du corps de manière binaire.
- Une autre solution proposée dans le cadre du projet BASE [CRL03] est d'identifier les parties des réponses qui vont contenir des éléments spécifiques aux serveurs et d'introduire des fonctions de comparaison de ces éléments : par exemple, si les serveurs sont synchronisés grâce à NTP, les dates des serveurs ne doivent pas différer de plus de quelques secondes.
- Une autre solution proposée dans [SDN03] est de considérer que tout le contenu dynamique est placé à un seul endroit, dans une base de données. L'architecture est modifiée pour prendre en compte ce fait par l'ajout d'un *adjudicator* qui sert de proxy/IDS pour les requêtes SQL. L'*adjudicator* compare également les requêtes SQL envoyées par chaque serveur COTS vers la base. La comparaison des corps peut donc être fait de manière binaire dans ce cas.

La solution du projet BASE est la plus générique puisqu'elle n'impose pas de conditions sur l'application qui fonctionne sur le serveur web. Elle peut cependant être difficile à mettre en place si l'application devient d'une taille imposante. Les deux autres solutions nécessitent de modifier l'application ou imposent des conditions sur l'application et l'ajout de composants (toutes les données dynamiques de l'application doivent être entreposées dans un conteneur unique accessible de tous les serveurs COTS). Dans nos tests sur des serveurs web dynamiques en 4.2.2.2, nous avons choisi de modifier l'application pour que les dates ne soient pas envoyées dans le corps des réponses.

Principe de l'algorithme Notre algorithme de détection repose sur trois parties : un chien de garde au niveau du proxy, le mécanisme de masquage (décrit dans la sous-section suivante) et l'algorithme de comparaison des réponses.

Un chien de garde permet de repérer les serveurs qui ne répondent pas à la requête. Les serveurs qui n'ont pas répondu sont considérés comme indisponibles et une alerte est levée pour chacun d'entre eux.

L'algorithme 1 détaille notre algorithme de comparaison. L'algorithme de comparaison est appliqué à l'ensemble des réponses reçues. Sont d'abord comparés le code de statut, ensuite les différents en-têtes choisis, puis le corps de la réponse. Lorsqu'aucune majorité parmi les réponses ne peut être trouvée, la comparaison est stoppée, l'IDS lève une alerte et une réponse d'erreur standard est envoyée au client. Si une majorité est trouvée, l'IDS indique au proxy la réponse à envoyer, qui est une réponse de la majorité. L'IDS lève une alerte pour chaque serveur ne faisant pas partie de la majorité. La comparaison des corps de réponse ne se fait que dans le cas où la réponse est de type 200, c'est-à-dire que la requête a été traitée avec succès par le serveur ; les autres types de réponses étant générés dynamiquement par les serveurs, la comparaison des corps de réponse causerait de nombreux faux positifs. S'il est possible de définir pour tous les serveurs les réponses envoyées en cas d'erreur, il est également possible de comparer le corps des réponses correspondant à des messages d'erreur des serveurs web.

Algorithm 1: Algorithme de détection par comparaison des réponses des serveurs web

Data: n le nombre de serveurs web et $R = \{R_i | 1 \quad i \quad n\}$ l'ensemble de réponses des serveurs web pour une requête donnée Req, D l'ensemble des différences de conception connues pour les serveurs web de l'architecture, $headers$ le tableau tel que $headers[0]$ = 'Content-Length', $headers[1]$ = 'Content-Type', $headers[2]$ = 'Last-Modified', $headers[3]$ = 'Location'

if $(Req, R) \in D$ **then**
  /* traiter les différences de conception qui ne sont pas dues à des vulnérabilités */
  modifier R pour masquer les différences

Partitionner R en C_i, $\forall (R_l, R_k) \in C_i^2, R_l.statusCode = R_k.statusCode$
$1 \quad i \quad m \quad n \qquad \forall (i,j) \in [1, m]^2 i \neq j, C_i \cap C_j = \emptyset$, m le nombre de partitions
if $\forall i \; Card(C_i) < n/2$ **then**
  Lever une alerte /* Pas de majorité dans l'ensemble des réponses */
  Sortir
else
  /* Une majorité existe dans l'ensemble des réponses */
  Trouver p tel que $Card(C_p) \quad n/2$
  if $C_p.statusCode \neq 2XX$ **then**
    /* Une majorité des serveurs web a renvoyée la même erreur : ce n'est pas la peine de comparer les autres en-têtes */
    for $k = 1$ *to* n **do**
      if $R_k \notin C_p$ **then** Lever une alerte pour le serveur k
    Sortir
  else
    /* Une majorité de serveurs a traité correctement la requête */
    for $i = 0$ *to* 3 **do**
      /* Comparer tous les en-têtes */
      $T \qquad C_p$ /* Affectation de l'ensemble contenant la majorité à T */
      Vider tous les C_j /* On efface toutes les partitions avant de créer les nouvelles */
      Partitioner T en C_j, $\forall (R_l, R_k) \in C_j^2, R_l.headers[i] = R_k.headers[i]$
      $1 \quad j \quad m \quad Card(T) \qquad \forall (j,k) \in [1, m]^2 j \neq k, C_j \cap C_k = \emptyset$, m le nombre de partitions
      if $\forall j \; Card(C_j) < n/2$ **then**
        Lever une alerte /* Pas de majorité dans l'ensemble des réponses */
        Sortir
      else
        /* Une majorité a été trouvée */
        Trouver p tel que $Card(C_p) \quad n/2$
    /* Comparer les corps des réponses */
    $T \qquad C_p$
    Vider tous les C_i
    Partition T in C_j, $\forall (R_l, R_k) \in C_j^2, R_l.body = R_k.body$
    $1 \quad j \quad m \quad Card(T) \qquad \forall (i,j) \in [1, m]^2 i \neq j, C_i \cap C_j = \emptyset$, m le nombre de partitions
    if $\forall j \; Card(C_j) < n/2$ **then**
      Lever une alerte /* Pas de majorité dans l'ensemble des réponses */
      Sortir
    else
      /* Une majorité a été trouvée */
      Trouver p tel que $Card(C_p) \quad n/2$
      for $k = 1$ *to* n **do**
        if $R_k \notin C_p$ **then**
          /* La réponse k est différente de la majorité */
          Lever une alerte pour le serveur k
      Sortir

3.2.3 Mécanismes de masquage

Le masquage des différences de spécification est réalisé à deux niveaux : au niveau du proxy par réécriture des requêtes et au niveau de l'IDS par modification des réponses. Comme expliqué en 2.1.4, ces mécanismes sont nécessaires pour limiter le nombre de faux positifs que pourrait générer l'algorithme de comparaison.

Normalisation des requêtes Le proxy est chargé de réécrire certaines requêtes pour les normaliser. Dans notre implémentation, nous n'avons pas choisi d'envoyer des requêtes éventuellement différentes à chaque COTS : tous les COTS reçoivent la même requête. Nous allons présenter trois règles de normalisation des requêtes mises en place dans notre prototype.

Sous Unix, la casse est prise en considération au niveau du système de fichiers, ce qui n'est pas le cas sous Windows. Une requête HTTP simple demandant l'envoi d'un document HTML peut aboutir à une erreur de type 404 (document non trouvé) si la casse ne correspond pas exactement à celle du fichier sous Unix, alors que sous Windows, le fichier serait trouvé et envoyé au client. Cette différence de comportement est due à une différence de spécification au niveau des systèmes d'exploitation utilisés. Le proxy est en charge de modifier les requêtes pour mettre en minuscule l'URL demandée ce qui permet de masquer ces différences, les noms de tous les fichiers accessibles sur nos serveurs étant en minuscule. Ceci pose une contrainte sur les fichiers accessibles par le serveur : si le serveur est statique, cette contrainte n'est pas très forte. Cela peut poser une contrainte sur les développeurs d'application web par contre.

Le serveur Apache gère correctement les requêtes contenant des ancres HTML (*anchors*) contrairement à d'autres serveurs. Ces ancres sont utilisées pour des liens internes à une page HTML. L'URL est alors suivi du caractère # et du nom de l'ancre. Les serveurs Abyss et thttpd, par exemple, ne gèrent pas ce type d'URL et renvoient une erreur 404 pour indiquer qu'ils n'ont pas trouvé le fichier correspondant à l'URL. Nous avons décidé de normaliser les requêtes en enlevant l'ancre de l'URL. Ainsi tous les serveurs sont capables de répondre au client de manière correcte. La petite différence du point de vue du navigateur est qu'il n'est pas redirigé sur l'ancre que l'utilisateur avait demandé.

Tous les serveurs ne gèrent pas l'en-tête *Range*. Cet en-tête permet au client de ne demander qu'une partie par de l'entité pointée par l'URL. Ceci permet, par exemple, de reprendre le téléchargement interrompu d'un fichier de taille importante sans avoir à télécharger de nouveau les données déjà téléchargées. Apache gère correctement cet en-tête ce qui n'est pas le cas de thttpd. En effet, thttpd envoie l'intégralité de l'entité. Nous avons décidé d'enlever cet en-tête de la requête. Le client reçoit alors l'intégralité de l'entité pointée par l'URL, ce qui ne correspond pas exactement à sa demande.

Modification des réponses Toutes les différences de spécification ne peuvent être masquées par la normalisation des requêtes. Au niveau de l'IDS, certaines

```
<request id="0">
  <description>requête sur un répertoire sans '/' 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 3.6 – Exemple d'une règle de masquage au niveau de l'IDS

réponses des serveurs sont modifiées avant l'exécution de l'algorithme de comparaison dans le but de masquer les différences de spécification qui n'ont pas été masquées par la normalisation des requêtes. La figure 3.6 présente un exemple de règle de masquage au niveau de l'IDS. Cette règle est exprimée en XML. Par rapport à la définition formelle d'une règle donnée en 2.1.4, les balises `regexpURI` et `regexpMethod` définissent par expression régulière l'ensemble d'entrées E. Les balises `server` définissent l'ensemble des réponses des serveurs S_i, c'est-à-dire les sorties attendues, produites par les différents serveurs confrontés à une entrée de E. Les autres balises : `compareContent`, `response`, `warn`, `alert` définissent l'ensemble d'actions A que doit effectuer l'IDS.

Une différence bénigne entre les codes de statut des réponses ne doit pas lever d'alertes. Ainsi, une requête HTTP de type GET dans laquelle l'URL pointe sur un répertoire du serveur mais ne se termine pas par un '/' entraîne des réponses différentes des serveurs utilisés dans les tests : certains serveurs renvoient un code 301, d'autres un code 302. Cette différence correspond bien à une différence de spécification et non à une intrusion. La Figure 3.6 donne la représentation en XML de la règle de masquage correspondant au cas explicité ci-dessus. Cette règle décrit la différence : si l'URL de la requête ne se termine pas par un '/' quelle que soit la méthode de la requête HTTP et si les serveurs

IIS et thttpd envoient une réponse HTTP dont le code de statut est 302 et l'en-tête « Content-Type » est text/html et le serveur Apache une réponse dont le code de statut est 301 et l'en-tête « Content-Type » est également text/html alors l'IDS transforme les réponses pour les rendre identiques à celle du serveur IIS, ne compare pas le corps de la réponse HTTP et ne lève pas d'alertes ni d'avertissements.

D'autres règles ont été écrites pour masquer des différences dans les en-têtes *Content-Type* quand les valeurs de celui-ci ne peuvent pas être spécifiés dans la configuration du serveur : ils sont inclus dans le code source du serveur thttpd ; il n'est donc pas possible de les modifier sans devoir recompiler le code du serveur et il semble plus naturel de les masquer au niveau de l'IDS. D'autres règles masquent des différences dans les codes de retour : par exemple, dans le cas où l'URL dépasse une certaine taille, le serveur IIS renvoie un code d'erreur 415 pour signaler que l'URL est trop longue alors que la plupart des autres serveurs, de manière générale, ne trouvent pas l'entité pointée par l'URL car elle n'existe pas et renvoie une erreur 404.

La combinaison de notre algorithme de comparaison et du masquage de différences de spécification permet d'obtenir des bons résultats en termes de détection que nous détaillons dans le chapitre suivant. Dans la suite, nous détaillons l'implémentation de notre approche de type boîte grise.

3.3 Approche boîte grise : description de l'implémentation

Nous décrivons dans cette section l'implémentation pour serveurs web de notre approche de type boîte grise. Nous présentons d'abord l'architecture de l'IDS. Pour obtenir les flux d'informations au niveau du système d'exploitation, nous récupérons les appels système. Nous décrivons donc ensuite la modélisation de ces appels système en termes de flux d'informations, puis comment nous construisons les graphes de flux d'informations. Nous discutons finalement les choix possibles pour les fonctions f et g utilisées dans le calcul de la similarité entre les graphes dans le cas des serveurs web.

3.3.1 Architecture

L'architecture détaillée de notre approche boîte grise est présentée sur la figure 3.7. Elle est composée de plusieurs éléments : un proxy HTTP, l'IDS fondé sur la similarité entre graphes de flux d'informations et, sur chaque serveur, un moniteur d'appels système, un générateur de graphes et un wrapper.

Le proxy HTTP a le rôle classique utilisé dans ce type d'architectures : il reçoit les requêtes provenant des clients et les transfère à tous les serveurs. Nous avons utilisé le même proxy que dans notre approche boîte noire. Celui-ci normalise certaines requêtes pour masquer des différences de spécification comme expliqué précédemment. Il attend ensuite leurs réponses. Idéalement, le

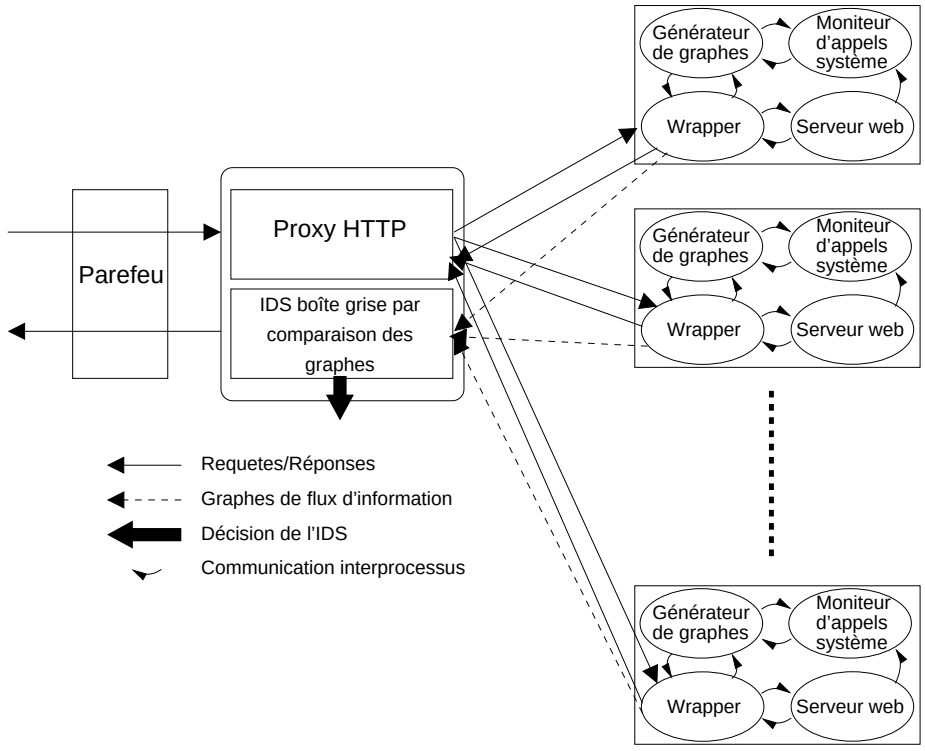


FIGURE 3.7 – Architecture de l'IDS par comparaison des graphes de flux d'informations

proxy attend la décision de l'IDS pour envoyer la réponse au client ; cependant, le temps de calcul de la similarité entre les graphes de flux d'informations est important. Il est préférable soit d'utiliser nos deux approches en conjonction, le proxy attend alors la décision de l'IDS boîte noire ; soit de renvoyer une des réponses reçues sans attendre la décision de l'IDS.

L'IDS reçoit les graphes de flux d'informations des différents wrappers, calcule les similarités entre ces graphes et lève une alerte dans les cas explicités dans la section 2.3.4.

Le rôle du moniteur d'appels système est d'enregistrer les appels système effectués par le serveur web et de les envoyer au générateur de graphes. Ces moniteurs d'appels système sont des programmes utilisateurs sur chacun des systèmes que nous avons utilisés : *strace* sous Linux, *ktrace* sous MacOS-X (*ktrace* fonctionne également sous FreeBSD, NetBSD et OpenBSD) et *detours* sous Windows. Ces trois programmes sont des processus utilisateurs ; des moniteurs d'appels système au niveau noyau seraient plus performants car ils généreraient moins de changements de contexte. Ils seraient cependant bien plus complexes à développer et maintenir.

Le générateur de graphes construit les graphes de flux d'informations à partir des appels système et les renvoie sur demande du wrapper. Nous détaillons son fonctionnement en 3.3.3.

Un problème reste à régler pour la construction des graphes de flux d'informations : comment associer les appels système à une requête donnée. Si la conception interne des serveurs COTS le permet, il est possible de faire l'association entre la requête et le processus qui traite cette requête. C'est le cas par exemple pour les serveurs Apache 1.3 et Abyss : un processus/thread ne traite qu'une seule requête à la fois. Il suffit alors d'ajouter, au niveau du proxy, un en-tête dans la requête contenant un identifiant unique qu'il sera possible de récupérer dans les logs d'appels système et ainsi faire l'association entre le processus et la requête. Ceci n'est pas toujours possible : le serveur thttpd n'utilise qu'un seul processus pour traiter toutes les requêtes : il n'est pas possible de réaliser l'association entre les appels système et une requête donnée par l'intermédiaire du processus exécutant ces appels système.

Deux possibilités sont envisageables pour régler le problème de l'association appels système/requêtes : n'utiliser que des serveurs où il est possible de faire cette association ou alors mettre en série les requêtes.

La première solution limiterait la diversité disponible et nécessiterait de vérifier pour chaque serveur COTS envisagé dans l'architecture si cette association est possible. C'est la solution choisie par Gao, Reiter et Song [GRS06b] : ils n'utilisent que deux COTS : un Apache sous Windows et un Apache sous Linux.

Comme il est déjà nécessaire de mettre en série les requêtes qui sont dépendantes, nous avons choisi la seconde solution. Les serveurs ne traitent alors qu'une seule requête à la fois. Le rôle du wrapper est de lier une requête particulière avec les temps de début et de fin de cette requête. Il reçoit une requête HTTP du proxy, stocke le temps de début de cette requête et la transfère au serveur. Il renvoie la réponse du serveur et, à la fin de cette réponse, stocke le temps de fin. Il demande ensuite le graphe de flux d'information correspondant au temps de début et de fin de la requête au générateur de graphes et envoie ce graphe à l'IDS. Il est ainsi possible d'obtenir la séquence d'appels système correspondant à une requête ; nous explicitons dans la suite comment il est possible d'obtenir les flux d'informations à partir de cette séquence d'appels système.

3.3.2 Modélisation des appels système

Pour obtenir les flux d'informations au niveau du système d'exploitation, le plus simple est de considérer les appels système effectués par un ou des processus car ces appels système sont l'interface entre les processus utilisateurs et le noyau du système d'exploitation. Il est éventuellement possible de les obtenir en observant chaque instruction assembleur exécutée par le processus mais les instructions assembleur réalisent des actions très limitées et éloignées sémantiquement des flux d'informations au niveau du système d'exploitation. Cette solution est donc beaucoup plus difficile à mettre en place. De plus, elle est beaucoup plus coûteuse en terme de performances.

Appels système	Modélisation en flux d'informations
read(fd,...) ;	(FILE|PIPE|SOCKET) -> PROCESS
readv(fd,...) ;	(FILE|PIPE|SOCKET) -> PROCESS
pread64(fd,...) ;	FILE -> PROCESS
recv(fd,...) ;	SOCKET -> PROCESS
recvfrom(fd,...) ;	SOCKET -> PROCESS
recvmsg(fd,...) ;	SOCKET -> PROCESS
write(fd,...) ;	PROCESS -> (FILE|PIPE|SOCKET)
writev(fd,...) ;	PROCESS -> (FILE|PIPE|SOCKET)
pwrite64(fd,...) ;	PROCESS -> FILE
send(fd,...) ;	PROCESS -> SOCKET
sendto(fd,...) ;	PROCESS -> SOCKET
sendmsg(fd,...) ;	PROCESS -> SOCKET
sendfile(fd1,fd2,...) ;	FILE2 -> PROCESS & PROCESS -> FILE1
sendfile64(fd1,fd2,...) ;	FILE2 -> PROCESS & PROCESS -> FILE1
mmap(„PROT_READ„fd,)	FILE -> PROCESS
mmap(„PROT_WRITE„fd,)	PROCESS -> FILE
mmap(„PROT_READ| PROT_WRITE„fd,)	PROCESS -> FILE & FILE -> PROCESS
mmap2(„PROT_READ„fd,)	FILE -> PROCESS
mmap2(„PROT_WRITE„fd,)	PROCESS -> FILE
mmap2(„PROT_READ| PROT_WRITE„fd,)	PROCESS -> FILE & FILE -> PROCESS
mprotect(„PROT_READ„,)	FILE -> PROCESS
mprotect(„PROT_WRITE„,)	PROCESS -> FILE
mprotect(„PROT_READ| PROT_WRITE„,)	PROCESS -> FILE & FILE -> PROCESS
fork()	PROCESS -> CHILD_PROCESS
vfork()	PROCESS -> CHILD_PROCESS
clone()	PROCESS -> CHILD_PROCESS & CHILD_PROCESS -> PROCESS

TABLE 3.1 – Modélisation des appels système Linux en termes de flux d'informations

Nous avons instrumenté environ 20 appels système sur chaque OS utilisé. Tous les appels système générant des flux d'informations, excepté trois que nous présentons dans la suite, correspondent au modèle des flux d'informations présenté dans la sous-section 2.3.1. Par exemple, un processus exécutant un appel système *read* est l'objet actif de l'opération de lecture du modèle ; le fichier, la socket ou le pipe lu est l'objet passif. Cet appel système correspond à une opération de lecture du modèle et crée donc un flux d'informations de

Appels système	Modélisation en flux d'informations
read(fd,...) ;	(FILE|PIPE|SOCKET) -> PROCESS
readv(fd,...) ;	(FILE|PIPE|SOCKET) -> PROCESS
aio_read(...) ;	(FILE|PIPE|SOCKET) -> PROCESS
pread(fd,...) ;	FILE -> PROCESS
recvfrom(fd,...) ;	SOCKET -> PROCESS
recvmsg(fd,...) ;	SOCKET -> PROCESS
write(fd,...) ;	PROCESS -> (FILE|PIPE|SOCKET)
writev(fd,...) ;	PROCESS -> (FILE|PIPE|SOCKET)
aio_write(...) ;	PROCESS -> (FILE|PIPE|SOCKET)
pwrite(fd,...) ;	PROCESS -> FILE
sendto(fd,...) ;	PROCESS -> SOCKET
sendmsg(fd,...) ;	PROCESS -> SOCKET
mmap(„PROT_READ„fd,)	FILE -> PROCESS
mmap(„PROT_WRITE„fd,)	PROCESS -> FILE
mmap(„PROT_READ| PROT_WRITE„fd,)	PROCESS -> FILE & FILE -> PROCESS
mprotect(„PROT_READ„,)	FILE -> PROCESS
mprotect(„PROT_WRITE„,)	PROCESS -> FILE
mprotect(„PROT_READ| PROT_WRITE„,)	PROCESS -> FILE & FILE -> PROCESS
fork()	PROCESS -> PROCESS_FILS
vfork()	PROCESS -> PROCESS_FILS

TABLE 3.2 – Modélisation des appels système MacOS-X en termes de flux d'informations

l'objet passif vers l'objet actif, étiqueté par les données lues, la date d'exécution et le type du flux d'informations, qui est la seule étiquette utilisée dans le calcul de la similarité entre les graphes. Les tableaux 3.1, 3.2 et 3.3 référencent les appels système créant des flux d'informations et leur modélisation en termes de flux d'informations sous Linux, MacOS-X et Windows. Dans ces tableaux, fd correspond au descripteur de fichier (*file descriptor*) du fichier, de la socket ou du pipe ; dans le cas de sendfile, fd1 correspond à FILE1 et fd2 à FILE2 ; PROCESS est le processus effectuant l'appel système ; CHILD_PROCESS est le processus fils créé par le processus appelant.

Les appels système qui créent un autre *thread* tel que *clone* (sur Linux) permettent aux processus père et fils de communiquer ensemble par l'intermédiaire de la mémoire. Il est possible de surveiller tous les flux de communication entre ces *threads* mais cela nécessite de vérifier tous les accès mémoire des processus père et fils, ce qui a un impact important sur les performances. Nous avons donc décidé de modéliser l'appel système *clone* par deux flux d'informations : un du père vers le fils et un du fils vers le père. Ces flux d'informations ont des étiquettes 'donnée' vides.

Appels système	Modélisation en flux d'informations
ReadFile(fd,...) ;	FILE -> PROCESS
ReadFileEx(fd,...) ;	FILE -> PROCESS
MapViewOfFile(fd,...) ;	FILE -> PROCESS
MapViewOfFileEx(fd,...) ;	FILE -> PROCESS
recv(fd,...) ;	(SOCKET|PIPE) -> PROCESS
recvfrom(fd,...) ;	(SOCKET|PIPE) -> PROCESS
WSARecv(fd,...) ;	(SOCKET|PIPE) -> PROCESS
WSARecvFrom(fd,...) ;	(SOCKET|PIPE) -> PROCESS
WriteFile(fd,...) ;	PROCESS -> FILE
WriteFileEx(fd,...) ;	PROCESS -> FILE
send(fd,...) ;	PROCESS -> (SOCKET|PIPE)
sendto(fd,...) ;	PROCESS -> (SOCKET|PIPE)
WSASend(fd,...) ;	PROCESS -> (SOCKET|PIPE)
WSASendTo(fd,...) ;	PROCESS -> (SOCKET|PIPE)
CreateProcess(...)	PROCESS -> PROCESS_FILS

TABLE 3.3 – Modélisation des appels système Windows en termes de flux d'informations

Les appels système créant un nouveau processus tels que *fork* sur Linux correspondent à une opération execute du modèle. Pour la même raison que précédemment, le flux d'informations créé par cet appel système a une étiquette 'donnée' vide.

Les appels système tels que *mmap* peuvent également être la source de flux d'informations qui ne peuvent être modélisés sans observer les accès mémoire des processus. L'appel système *mmap* permet à un processus de projeter en mémoire un fichier. Lire ou écrire dans le fichier se fait tout simplement par des lectures ou des écritures en mémoire. Suivant les argument de *mmap*, nous avons décidé de créer des flux d'informations entre le processus qui exécute cet appel système et le fichier considéré. Si la projection est en lecture seule (resp. écriture seule), un flux d'informations du fichier (resp. du processus) vers le processus (resp. le fichier) est créé. Si la projection est en mode lecture-écriture, deux flux d'informations sont créés du processus vers le fichier et du fichier vers le processus. Ces flux d'informations ont également des étiquettes 'donnée' vides.

Notre prototype ne prend pas en compte certains mécanismes d'IPC tels que les messages et les mémoires partagées et les signaux. Les mécanismes d'IPC peuvent être modélisés par des objets passifs. Les signaux créent des flux d'informations entre le processus qui envoie le signal et celui qui le reçoit. Il est donc possible de les modéliser par un flux d'informations avec une étiquette 'donnée' qui contient le numéro du signal envoyé.

La granularité offerte par les appels système n'est pas suffisante si l'on veut obtenir précisément les flux d'informations entre les différents objets du système d'exploitation : la communication entre *threads* par accès direct à la mémoire, ainsi que les accès à un fichier projeté en mémoire, ne sont pas entièrement

réalisés à travers des appels système. Pour suivre ces flux d'informations, il est nécessaire de surveiller les accès mémoire des processus à des adresses particulières, qui correspondent aux données partagées entre les *threads* ou à l'adresse où est projeté le fichier en mémoire par exemple. Techniquement, il n'est pas évident de faire ceci de manière performante.

À partir cette modélisation, il est possible de construire les graphes de flux d'informations correspondant aux requêtes.

3.3.3 Construction des graphes de flux d'informations

Nous détaillons ici quelques propriétés de la construction de graphes de flux d'informations à partir des appels système.

La construction des graphes nécessite d'instrumenter effectivement d'autres appels système que ceux générant des flux d'informations présentés dans la sous-section précédente. Il est également nécessaire d'instrumenter tous les appels système créant des objets passifs tels que *open*, *socket*, *accept*, *pipe* sous MacOS-X par exemple, les appels système modifiant certaines informations comme *dup*, *dup2* ou *fcntl* qui modifient un descripteur de fichier associé à un fichier, une socket ou un pipe par exemple, ou encore *execve* qui modifie un objet processus. Les appels système *close*, *shutdown* ou *exit* doivent également être instrumentés car ils modifient certains éléments permettant de suivre les flux d'informations du système.

Pour masquer certaines différences de conception entre les COTS au niveau des flux d'informations, il est nécessaire de regrouper toutes les écritures d'un même processus vers un même objet passif, ainsi que toutes les lectures d'un même objet passif par un processus. En effet, que le processus lise le contenu d'un fichier octet par octet ou kilo-octet par kilo-octet, en termes de flux d'informations, cela n'a pas d'importance. Tous ces flux d'informations sont regroupés ensemble et les données lues sont concaténées dans l'étiquette du flux d'informations résultant.

Dans le même but que celui du regroupement des différentes lectures ou écritures, lors de la construction du graphe, nous essayons de filtrer les pipes lorsque cela est possible : nous remplaçons l'écriture de données dans un pipe puis la lecture par un autre processus des mêmes données par un flux d'informations unique entre les deux processus avec une étiquette 'donnée' contenant les données transférées entre les deux processus. S'il n'y a pas de données en attente dans le pipe (toutes les données écrites ont été lues), l'objet pipe est éliminé du graphe.

3.3.4 Similarité entre les graphes de flux

Nous détaillons une optimisation supplémentaire introduite dans le calcul de la similarité et les choix des fonctions f et g que nous avons pris dans le cadre des serveurs web.

Optimisations du calcul En plus de l'interdiction de certains mappings présentée en 2.3.3.2, dans le cadre des serveurs, nous avons décidé d'interdire d'autres mappings. Ceci permet d'éviter des associations qui n'ont pas de sens et permet également d'optimiser le calcul. Du point de vue du serveur web, le système de fichiers peut être partitionné en deux : l'espace du serveur web et le reste. L'espace du serveur web contient les fichiers qui sont accessibles par l'intermédiaire du serveur web et le reste, les fichiers qui ne le sont pas. Les fichiers qui sont à l'intérieur de l'espace web ne peuvent être associés dans un mapping unitaire à des fichiers en dehors de l'espace web. De plus, les fichiers qui sont à l'intérieur de l'espace web ne peuvent pas être associés à des fichiers de l'espace web qui ont un nom différent.

Choix des fonctions f et g Ces fonctions ont un impact important sur le calcul de la similarité. Il faut les choisir convenablement pour refléter la signification des mappings effectués lors du calcul de la similarité. Dans notre prototype, nous avons testé deux fonctions différentes pour f et une pour g. D'abord, nous avons choisi pour f et g le cardinal des ensembles considérés. Ensuite, nous avons modifié f sans changer g. En utilisant les mêmes notations que dans la sous-section 2.3.3, on considère $S \in V_1 \cup V_2 \cup E_1 \cup E_2$ et $(v, v', l) \in r_{E_1} \cup r_{E_2}$. f est défini par :

$$\begin{aligned} f(\emptyset) &= 0 \\ f(S \cup \{v, l\}) &= f(S) + 1 \\ f(S \cup \{(v, v', l\}) &= f(S) + 1 \ \ si \ \ l \neq PROCESS\_TO\_FILE \\ f(S \cup \{(v, v', l\}) &= f(S) + 3 \ \ si \ \ l = PROCESS\_TO\_FILE \end{aligned}$$

Avec cette fonction f, si un flux d'informations de type PROCESS_TO_FILE n'est pas associé dans le mapping m, cela va entraîner une baisse plus importante de la similarité entre les deux graphes considérés. Cette fonction permet de mettre l'accent sur la détection des attaques contre l'intégrité du serveur.

3.4 Combinaison des deux approches

Il est possible de combiner les deux approches comme proposé en 2.4. Pour la corrélation entre les deux IDS, nous avons utilisé la même logique que celle décrite dans la section susnommée, c'est-à-dire qu'une alerte est levée si un des IDS lèvent une alerte. Au niveau de l'architecture, étant donné le temps de calcul des similarités, nous avons choisi de placer l'IDS de type boîte noire sur le chemin de traitement de la requête, c'est-à-dire que le proxy attend la réponse de l'IDS de type boîte noire pour renvoyer la réponse au client. L'IDS de type boîte grise n'est pas, quant à lui, placé sur le chemin de la requête, c'est-à-dire que le proxy n'attend pas sa décision pour envoyer une réponse au client.

Nous avons également mis en place une collaboration entre nos deux IDS. Quand les deux IDS sont utilisés conjointement, l'IDS de type boîte grise utilise la décision de l'IDS de type boîte noire dans son calcul de mapping. L'IDS de

type boîte noire compare, en effet, les réponses des serveurs, ce qui correspond dans la modélisation par flux d'information au niveau du système d'exploitation aux flux en écriture sur la socket qui correspond au client. Nous utilisons la décision de l'IDS de type boîte noire pour déterminer si ces flux sortant sont considérés comme associés ou non. Cette implémentation permet de raffiner le calcul de similarité en prenant en compte les données d'un seul flux : celui d'écriture sur la socket du côté client. Si on veut prendre en compte cette modification dans le modèle, on considère que l'IDS de type boîte grise change l'étiquette de ce flux d'écriture en fonction du résultat de l'IDS de type boîte noire. Dans le cas où $N = 3$, si les réponses des serveurs S_1 et S_2 sont considérées équivalentes par l'IDS de type boîte noire et sont différentes de la réponse du serveur S_3, alors les étiquettes des flux d'écriture sur la socket qui correspond au client dans les graphes de S_1 et S_2 sont modifiées en $PROCESS\_TO\_SOCKET\_12$ et celle dans le graphe de S_3 est modifiée en $PROCESS\_TO\_SOCKET\_3$. Ceci fait que l'on peut considérer que les flux de S_1 et S_2 sont associés ensemble contrairement aux mappings entre S_1 et S_3 et entre S_2 et S_3.

Nous avons présenté dans ces sections comment était appliqué au cas des serveurs web le modèle général décrit dans le chapitre précédent pour les deux approches proposées. Ces deux approches permettent de détecter les intrusions affectant le serveur web car celui-ci est diversifié. Elles ne permettent pas de détecter toutes les intrusions affectant éventuellement l'application web fonctionnant au-dessus du serveur web. Nous proposons dans la section suivante une méthode permettant d'augmenter le taux de couverture de notre approche.

3.5 Diversification automatique des scripts web : randomisation des éléments du langage

Nous présentons dans cette section une solution pour augmenter le taux de couverture de nos approches dans le cadre des serveurs web. Nous explicitons, d'abord, les problèmes liés à la diversification d'applications web dans les approches de détection d'intrusions par diversification de COTS. Ensuite, nous détaillons la solution que nous proposons. Nous n'avons pas d'implémentation automatique de cette méthode.

3.5.1 Diversification des applications web

Si les applications web fonctionnant au-dessus des serveurs web ne sont pas diversifiés, l'hypothèse d'indépendance d'exploitation des vulnérabilités n'est plus assurée. Il est alors possible de ne pas détecter certaines intrusions affectant l'application web elle-même. Les intrusions qui sont indépendantes du sous-système (serveur web, système d'exploitation, matériel) sur lequel fonctionne l'application ne pourront pas être détectées. Les intrusions qui dépendent de ces paramètres peuvent être détectées grâce à la diversification des sous-systèmes. Par exemple, une injection de code binaire, par l'intermédiaire d'un

buffer overflow dans un script CGI écrit en C ou C++, n'est pas indépendante du processeur et éventuellement du système d'exploitation (en cas, d'appels système notamment). Cette injection de code pourra donc être détectée. Par contre, il ne sera pas possible de détecter une injection de code PHP par l'intermédiaire d'une vulnérabilité de type *include* si le code injecté n'est pas spécifique au serveur web ou au système d'exploitation. Ce problème touche toutes les approches présentées dans la section 1.3.

Il semble donc nécessaire de diversifier les applications si l'on veut augmenter le taux de couverture. Les méthodes de diversification par programmation de versions ou par l'utilisation de COTS posent cependant des problèmes dans le cadre des applications web.

3.5.1.1 Problématique

Pour augmenter la couverture de détection de l'approche par diversification de COTS, il serait nécessaire de diversifier également les applications web. Il faut distinguer deux cas : soit l'application web est elle-même un COTS, soit c'est une application web développée pour un besoin interne spécifique.

Dans le premier cas, de nombreux COTS existent pour remplir différentes fonctions : serveurs d'applications en java (Tomcat, Websphere, Jonas, etc.) ou un forum (phpbb, vBulletin, Phorum, etc.). La diversification de COTS n'est cependant pas vraiment possible à ce niveau, les spécifications des produits étant complètement différentes : une requête HTTP permettant l'envoi d'un message sur un forum phpbb n'a pas du tout le même format que celui d'une requête pour un forum vBulletin. On pourrait penser à développer un proxy permettant de convertir les requêtes pour un COTS donné en requêtes pour les autres mais cela semble être un travail important, très spécifique et voire dans certains cas, impossible.

Dans le second cas, il est alors nécessaire de développer plusieurs fois l'application de manière différente : c'est de la programmation N-versions. Cependant si l'on ne souhaite pas modifier nos algorithmes de comparaison, il faut spécifier exactement les sorties réseau de l'application et les flux d'informations au niveau du système d'exploitation, en plus de la spécification fonctionnelle qui se doit d'être précise au niveau de l'interface. Cette méthode présente trois défauts : la nécessité de développer plusieurs fois la même application, ce qui engendre un coût important, la définition précise de l'interface (méthode HTTP pour l'envoi des formulaires GET ou POST, ordre et nombre des paramètres, par exemple ; ces choix étant laissés à l'appréciation du développeur généralement) et la définition des spécifications non-fonctionnelles. Le troisième défaut est spécifique au choix des algorithmes de comparaison. Toutes les approches par diversification de COTS ne sont pas forcément affectées : l'approche suivie dans HACQIT [RJCM03] ou celle de Gao, Reiter et Song [GRS06a] évitent cet écueil. En effet, dans HACQIT, seuls les codes de statut des réponses HTTP sont comparés. La comparaison des comportements dans les travaux de Gao, Reiter et Song repose sur un apprentissage préalable des correspondances.

Il nous semble difficile de diversifier soit par l'intermédiaire de COTS, soit

par programmation d'autres versions. Les COTS posent un réel problème au niveau de l'interface. La programmation de versions a un coût important et nécessite de définir une spécification précise de l'interface qui est généralement laissée à l'appréciation du développeur web. Notre solution partielle repose sur la diversification automatique.

3.5.1.2 Diversification automatique

La solution que nous proposons est d'utiliser des techniques de diversification automatique comme celles présentées par Kc, Keromytis et Prevelakis [KKP03]. Les méthodes présentées dans le cadre du projet DieHard [BZ07], N-variant [NTEK+08] ou par Babak Salamat [SGF08] peuvent être utilisées par les applications web codées en C ou C++. Elles permettent, en effet, de détecter des attaques bas-niveau (injection de code binaire). Nous proposons une méthode permettant de détecter des injections de code de plus haut niveau dans les langages de script qui sont utilisés couramment dans le développement d'applications web : perl, php, python ou ruby, par exemple. Ces langages sont énormément utilisés dans les applications web de nos jours, bien plus que le C ou le C++.

Notre méthode repose sur les travaux de Kc, Keromytis et Prevelakis [KKP03] : ils proposent de modifier l'interpréteur des langages de haut-niveau grâce à une graine aléatoire. L'interpréteur, ainsi modifié, comprend un autre langage que le langage de base. La figure 3.8 explicite leur méthode sur un exemple simple de scripts perl. Les opérateurs, les appels de fonctions et les mots clés du langage sont modifiés par l'ajout d'un tag. L'interpréteur ne comprend plus le langage de base mais le langage modifié par l'ajout du tag. Si l'attaquant veut injecter du code qui sera interprété correctement, il lui faut connaître la graine aléatoire. Il est toujours possible à l'attaquant de chercher cette graine par attaque de type *brute force* ou par des méthodes plus évoluées [SEP05] (bien que la technique présentée dans cet article ne fonctionne pas pour les langages de haut-niveau). La probabilité que l'attaquant trouve la graine est faible si l'aléa est suffisamment important.

Notre méthode repose sur l'utilisation de cette technique dans le cadre de notre architecture : l'application web reste la même sur les différents serveurs COTS, elle est diversifiée sur chaque serveur avec une graine différente et les interpréteurs sont modifiés pour interpréter le langage modifié. Même si l'attaquant connaît toutes les graines utilisées, il ne pourra injecter du code qui sera interprété par tous les serveurs web. En cas d'injection de code, au maximum un seul interpréteur va interpréter le code injecté ; les autres vont se terminer par une erreur. Il est ainsi possible de détecter l'intrusion en comparant les codes de retour des interpréteurs. Trois cas sont possibles :

- Tous les interpréteurs ont interprété correctement le script : il n'y a pas eu d'intrusion par injection de code.
- Un interpréteur a correctement interprété le script, les autres ont échoué. Il y a eu une intrusion effective sur le serveur dont l'interpréteur a réussi. Une alerte doit être levée ; le serveur compromis doit être reconfiguré.

```
foreach $k (sort keys %$tre)
{
  $v = $tre->{$k};
  die ''duplicate key $k\n''
                if defined $list{$k};
  push @list, @{ $list{$k} };
}

foreach123456789 $k (sort123456789 keys %$tre)
{
  $v =1234567889 $tre->{$k};
  die123456789 ''duplicate key $k\n''
                if123456789 defined123456789 $list{$k};
  push123456789 @list, @{ $list{$k} };
}
```

FIGURE 3.8 – Transformation d'une partie d'un programme perl par ajout d'un tag

– Tous les interpréteurs ont échoué dans l'interprétation du script : il y a eu une tentative d'injection de code. Du code intrusif a cependant pu être exécuté sur un des serveurs. Une alerte doit être levée et une analyse plus poussée de l'attaque doit être menée pour déterminer si oui ou non, un serveur a pu être compromis.

La détection de l'injection n'est pas probabiliste dans notre cas : elle est assurée. Il faut noter que cette méthode permet d'étendre la couverture de détection aux intrusions qui consistent en l'injection de code haut-niveau. Cependant les intrusions n'injectant pas de code ne seront pas, de manière générale, détectées.

Le modification de l'interpréteur n'est pas un problème en soi. La modification des scripts, par contre, peut être une tâche plus ardue car ceux-ci peuvent générer d'autres types de code : un script php peut, par exemple, générer du « code » shell ou des requêtes SQL. Si l'on souhaite aussi détecter les injections de code généré par le script, la tâche est plus difficile. On examine, dans la suite, le problème du point de vue du langage racine (le premier à s'exécuter et qui génère possiblement d'autres types de code) et du point de vue des langages générés.

3.5.2 Randomisation du langage racine

La randomisation des fichiers source est aisée à faire automatiquement d'après [KKP03]. Il suffit de remplacer les opérateurs et mots clés choisis et les appels de fonction par leur remplaçant qui peuvent par exemple consister à ajouter un

nombre aléatoire ou tout autre symbole ou suite de symboles. Il n'est même pas nécessaire que le nombre aléatoire ou la suite de symboles soit de taille importante, étant donné que l'idée est que les langages soient distincts sur chaque serveur. Ceci est possible de manière automatique en modifiant *perltidy* comme proposé dans [KKP03]. L'inclusion de code externe par l'intermédiaire de modules oblige également à modifier le code de tous les modules utilisés dans l'application.

Il faut également modifier l'interpréteur pour que celui-ci comprenne le nouveau langage choisi. Deux solutions sont possibles : modifier directement l'interpréteur ou utiliser un programme qui va « dérandomiser » le programme puis le passer à l'interpréteur non-modifié. La première solution a été celle mise en place dans [KKP03], la seconde dans [BK04] où la technique proposée dans l'article de Kc, Keromytis et Prevelakis est appliquée au langage SQL. La seconde méthode est légèrement moins performante mais également moins intrusive puisqu'il n'est pas nécessaire de modifier l'interpréteur.

Au niveau du langage racine, il n'y a pas de difficultés pour modifier l'interpréteur ou les scripts.

3.5.3 Randomisation des langages générés

Il est bien plus difficile de « randomiser » les éléments de langages générés que ceux du langage racine. En effet, le code généré peut l'être de manière dynamique et sera généralement stocké dans des chaînes de caractères dans le langage racine. La figure 3.9 présente quelques difficultés qu'il est possible de rencontrer dans le traitement des éléments des langages générés. Cet exemple prend l'exemple d'un script php générant une requête SQL : les mots clés peuvent être stockés dans des variables, peuvent être concaténés dynamiquement, les variables peuvent être utilisées dans des circonstances non reliées à la requête SQL, etc. Cet exemple est irréaliste mais présente certaines des difficultés pouvant éventuellement survenir.

Plusieurs méthodes peuvent être envisagées pour résoudre ce problème. L'analyse statique de code et des dépendances entre les variables doit permettre de résoudre certains cas et identifier les éléments du langage que l'on souhaite « randomiser » parmi les chaînes de caractères utilisées dans le script. De l'analyse dynamique peut être utilisée dans une phase d'apprentissage mais la modification dynamique est à proscrire puisqu'elle rendrait l'approche inopérante en cas de code intrusif injecté. Une autre méthode est d'imposer au développeur des contraintes sur son style de programmation soit en permettant au programme « randomisant » de repérer aisément les éléments des langages générés soit en développant de manière stricte pour que de l'analyse statique puisse fonctionner.

3.5.4 Limitations

Cette méthode, malgré ses avantages en termes de coût, de facilité de mise en place dans le cas général, présente quelques limitations.

Il est nécessaire de disposer du code source des scripts de l'application et des modules utilisés pour pouvoir les modifier. Les langages de script sont gé-

```
<?php

...

$fr = "FR";
$om = "OM";
$where = "WHERE";
$requete = "SEL" . "ECT * " . $fr . $om . "users " . $where
                                        . "login=" . $login . ";";

$site = $where . "is Mike?";

$resultat =  mysql_query($requete);

...

?>
```

FIGURE 3.9 – Exemples de difficultés dans la « randomisation » automatique des langages générés

néralement interprétés et le code est donc disponible. Dans d'autres cas, il est cependant possible, par exemple, d'avoir des applications en python compilé, le code source n'étant alors pas disponible.

L'autre limitation de cette approche est qu'elle se limite aux applications web dans lesquelles le client n'envoie pas de code directement à l'application. Une application web qui permet à l'utilisateur de rentrer ses propres requêtes SQL (dans un but d'apprentissage par exemple) ne peut pas être diversifiée au niveau du langage SQL. Une grande partie des applications web ne nécessite cependant pas de recevoir du code provenant du client.

3.6 Résumé

Dans ce chapitre, nous avons présenté la manière dont nous avons appliqué le modèle général de détection d'intrusions au cadre des serveurs web. Nous avons décrit plus précisément le protocole HTTP qui est un protocole léger et universel. Ses caractéristiques permettent d'expliquer les quelques difficultés que doit résoudre notre algorithme de comparaison des réponses réseau. Nous avons décrit précisément cet algorithme de comparaison et détaillé les mécanismes de masquage des différences de spécification que nous avons mis en place : normalisation des requêtes et masquage des différences de sorties connues.

Nous avons présenté également notre IDS par comparaison des graphes de

flux d'informations. Les flux d'informations sont obtenus grâce à l'interception des appels système. Ces appels système ne sont cependant pas suffisants pour obtenir tous les flux d'informations. Les autres méthodes pour les obtenir semblent entraîner une baisse de performance trop importante cependant. Nous avons détaillé le processus de création des graphes de flux d'informations et exposé quelques choix concernant le calcul de la similarité entre les graphes. Nous avons brièvement présenté notre architecture combinant les deux IDS.

Finalement, nous avons exposé le problème posé par l'exécution d'applications web au-dessus des serveurs web et une solution pour résoudre partiellement ce problème, solution qui permet de détecter les injections de code de haut-niveau.

Chapitre 4

Tests et résultats de détection, diagnostic et performances

« *Half the people you know are below average.* »
Steven Wright

Ce chapitre décrit les tests que nous avons menés et les résultats que nous avons obtenus avec les prototypes d'IDS que nous avons développés pour les serveurs web. Nous allons tout d'abord décrire l'environnement de test que nous avons utilisés et les sources de données. Ensuite, nous présentons les tests et les résultats obtenus avec notre prototype d'IDS de type boîte noire. Nous faisons de même pour notre prototype d'IDS de type boîte grise puis étudions les résultats de leur combinaison. Finalement, nous nous attachons à décrire quelques expériences et résultats obtenus par notre approche de diagnostic des anomalies et les performances de nos prototypes.

4.1 Description de l'environnement et des sources de données

Tous nos tests ont été menés en environnement contrôlé en utilisant, dans certains cas, des logs issus d'un environnement réel. Nous n'avons pas eu l'occasion de faire des tests en environnements réels. Il est en effet nécessaire de mettre en place les différents serveurs COTS et le proxy. Notre approche de détection ne consiste pas seulement en l'ajout d'un IDS mais bien en la modification du serveur, remplacé par notre architecture de détection, ce qui rend la mise en place de tests en environnements réels bien plus difficile.

4.1.1 Description de l'environnement contrôlé

Nous avons utilisés deux types de serveurs pour nos différents tests en environnement contrôlé : un serveur web statique et un serveur web dynamique.

Le contenu du serveur web statique consiste en une copie du contenu du serveur web du campus de Rennes de Supélec (`http://www.rennes.supelec.fr/ren/`) sans les parties dynamiques comme la recherche, par exemple. Nous avons utilisé deux versions différentes du contenu : la première version date de mars 2005 et a été récupérée grâce à un script ; la seconde version date d'un backup du 10 avril 2006.

Le contenu du serveur web dynamique consiste en une image d'une partie d'un serveur web interne à l'équipe SSIR. La partie copiée concerne les données gérées par l'application php Bibtex Manager (`http://www.rennes.supelec.fr/ren/perso/etotel/PhpBibtexDbMng/`) et l'application en elle-même. Cette application permet le partage de références bibliographiques dans une équipe de recherche. L'application est composée de scripts écrits en php ; les données gérées par l'application sont une base de données MySQL et des fichiers contenant les articles référencés dans la base MySQL. La version de l'application et des données datent d'avril 2006.

Pour tester nos IDS dans ces deux environnements, il nous faut également du trafic HTTP à destination de ces deux serveurs. Pour cela, nous avons eu accès à deux types de données : les logs du serveur web du campus de Rennes de Supélec et du trafic HTTP à destination d'une copie du serveur web interne à l'équipe. Ces requêtes sont, dans la majorité, non-intrusives : il est difficile de savoir avec exactitude s'il y a des requêtes conduisant à des intrusions vu la quantité de données représentées par ces logs ou le trafic sauvegardé. Ces ensembles vont nous permettre de tester la pertinence de nos IDS. Pour tester leur précision, nous avons développé des attaques permettant d'exploiter des vulnérabilités.

4.1.2 Logs du serveur web

Nous avons eu accès à certains fichiers de log du serveur web du campus de Rennes de Supélec : une première partie couvre le mois de mars 2003, la seconde partie couvrant, quant à elle, la période du 3 au 10 septembre 2006 et du 15 au 22 octobre 2006.

Les logs se présentent sous la forme d'un fichier texte où chaque entrée correspond à une requête. Le format CLF (*common log format*) est le format standard pour stocker les logs HTTP. La ligne ci-dessous représente une entrée dans ce format :

```
1.1.1.1 - - [20/Feb/2005:04:15:30 +0100]
                        "GET /ren/biblio/liens.htm HTTP/1.0" 200 9754
```

Ce format présente plusieurs champs : l'adresse IP du client, des informations d'identification (ici, non présentes et remplacées par des '-'), la date de la requête, la première ligne de la requête, le code de statut de la réponse et le

nombre d'octets envoyés dans la réponse du serveur web. Ce qui nous intéresse ici est la première ligne de la requête, celle-ci comporte la majeure partie de la sémantique de la requête : la méthode et l'URI de la ressource demandée. Cette première ligne est suffisante dans le cadre d'un serveur web statique pour être rejouée sans autre information dans le cas des protocoles HTTP/0.9 et HTTP/1.0. Le protocole HTTP/1.1 oblige en effet à préciser dans la requête l'en-tête *Host*. Cet en-tête est utile pour les serveurs physiques hébergeant plusieurs sites web de manière transparente. Nous l'avons fixé à l'adresse IP du proxy dans le cadre de nos tests.

Si ces logs sont suffisants pour rejouer les requêtes sur un serveur web statique, ce n'est plus le cas pour un serveur web dynamique : le client peut envoyer des données par l'intermédiaire d'une requête POST, les données étant alors situées dans le corps de la requête, les éléments d'identification comme les cookies sont stockés dans un en-tête de la requête, etc. Utiliser les fichiers de logs du serveur web n'a donc pas de sens dans le cadre d'un serveur web dynamique. Nous avons donc capturé du trafic HTTP complet pour obtenir des requêtes plus réalistes.

4.1.3 Trafic provenant d'une utilisation artificielle

Il ne nous a pas été matériellement possible de sauvegarder du trafic à destination du serveur interne de notre équipe, sur lequel fonctionne l'application web php Bibtex Manager. Pour obtenir du trafic normal à destination de cette application, nous avons configuré un autre serveur sur lequel nous avons installé php Bibtex Manager et avons demandé aux personnes de l'équipe d'utiliser ce serveur en imitant leur comportement habituel sur le serveur réel interne à l'équipe. Nous avons ainsi sauvegardé du trafic à destination d'un clone du serveur, représentatif d'un trafic normal sur le serveur.

Les requêtes obtenues grâce aux logs de serveur et le trafic sauvegardé est a priori du trafic normal au moins en grande partie. Ces requêtes pourront être utilisées pour tester la pertinence de nos prototypes d'IDS. Pour tester la fiabilité des IDS, nous avons développé ou utilisé quelques attaques à l'encontre des serveurs web ou de l'application php Bibtex Manager.

4.1.4 Descriptif des attaques

Nous décrivons tout d'abord les attaques utilisées ou développées contre les serveurs HTTP. Ces attaques se concentrent sur l'exploitation de vulnérabilités dans les serveurs COTS eux-mêmes. À l'opposé, dans le cas du serveur web dynamique, les vulnérabilités ont été introduites dans l'application web php Bibtex Manager.

4.1.4.1 Attaques et vulnérabilités des serveurs web

Nous avons exploité plusieurs vulnérabilités dans différents serveurs : un serveur HTTP minimal, Apache, IIS et thttpd.

Vulnérabilités du serveur HTTP minimal Ce serveur offre un service HTTP minimal. Il comporte un grand nombre de vulnérabilités qui peuvent être facilement exploitées. Néanmoins, ces attaques constituent un prototype des attaques typiques que l'on peut trouver à l'encontre de serveurs tels qu'Apache ou IIS. Nous avons modifié le serveur de telle manière qu'il se conforme mieux aux spécifications du protocole HTTP/1.0, en envoyant des en-têtes et notamment ceux qui nous servent dans notre algorithme de comparaison de contenu.

Nous avons exploité plusieurs vulnérabilités du serveur HTTP minimal, de manière à produire les effets suivants :

1. Accès à des fichiers soit faisant partie du site, soit des fichiers confidentiels ;
2. Modification de fichiers ;
3. Exécution d'opérations locales sur le serveur ;

La première vulnérabilité exploitée concerne les possibilités de remontées dans l'arborescence pour accéder à des fichiers confidentiels, dans le sens non-publics, c'est-à-dire qu'ils ne sont pas dans l'arborescence normalement servie par le serveur web. Le serveur HTTP minimal n'effectue aucun contrôle des chemins d'accès dans les URL. Il est donc possible de l'utiliser pour accéder au contenu de fichiers lisibles uniquement par root, par exemple au moyen de la requête :

```
GET /../../../../../../etc/shadow HTTP/1.0
```

En réponse à cette requête, ce serveur envoie le document demandé, c'est-à-dire le fichier */etc/shadow*. Ce fichier contient le *hash* des mots de passe de la machine.

La deuxième vulnérabilité exploitée est la possibilité de remonter dans l'arborescence à partir du répertoire d'exécution des scripts CGI pour exécuter n'importe quel programme.

```
GET /cgi-bin/../../bin/sh
```

La requête suivante permet de modifier le fichier */etc/shadow* du serveur :

```
GET /cgi-bin/../../bin/sh -c 'echo root::12492::::: > /etc/shadow'
```

Le fichier */etc/shadow* est modifié et ne contient plus qu'une seule ligne indiquant que le compte administrateur (*root*) de la machine n'a pas de mot de passe.

La troisième vulnérabilité exploitée est un débordement de tampon présent dans le traitement de l'URL. En effet, le serveur HTTP minimal ne vérifie pas la taille de l'URL dans les requêtes et utilise un tampon de taille fixe (512 octets) pour stocker la première ligne de la requête. Il est alors possible de faire

exécuter n'importe quelles instructions par le processeur. Nous avons également exploité cette vulnérabilité pour effectuer un déni de service. Pour cela, nous avons modifié le serveur HTTP afin qu'il utilise une approche fondée sur l'appel *select* plutôt qu'une approche de type *fork* pour gérer les connexions réseau. Il est alors très simple de faire défaillir le processus du serveur et donc de réaliser une attaque contre la disponibilité du serveur.

Vulnérabilité affectant Apache Nous avons exploité une vulnérabilité contre Apache pour attaquer la confidentialité du serveur. Cette vulnérabilité (Bugtraq ID 2503) consiste en l'envoi d'un grand nombre de caractères '/' dans la requête. Si le serveur Apache a les modules : mod_dir, mod_negociation, mod_autoindex activés, il est alors vulnérable à cette attaque et envoie la liste des fichiers du répertoire de la requête. Une requête du type :

```
GET //////////////...///// HTTP/1.0
```

permet d'obtenir la liste des fichiers du répertoire racine servi par le serveur. Les versions d'Apache précédant la version 1.3.19 sont vulnérables.

Vulnérabilité affectant IIS Nous avons exploité une vulnérabilité affectant IIS. Cette vulnérabilité (CVE-2000-0884) permet, suivant la charge utile, d'affecter la confidentialité, l'intégrité ou la disponibilité du serveur. IIS 4.0 et 5.0 permettent à l'attaquant de lire des documents à l'extérieur de l'arborescence servie par le serveur web et/ou d'exécuter des commandes arbitraires, par l'intermédiaire d'URL malformées contenant des caractères encodés en unicode et permettant une remontée dans l'arborescence. La requête suivante permet d'obtenir le contenu du répertoire courant par l'intermédiaire de l'exécution du shell cmd.exe et de la commande dir :

```
GET /scripts/..%c1%1c../winnt/system32/cmd.exe?/c+dir HTTP/1.0
```

Vulnérabilité affectant thttpd Nous avons mis en œuvre une vulnérabilité contre le serveur thttpd. Cette vulnérabilité est de type cross site scripting (Bugtraq ID 4601). Elle s'applique notamment à la version 2.19 du serveur. Le serveur récupère sans traitement l'URL de la requête pour l'afficher dans sa réponse. Il est ainsi possible d'appeler un document en référençant un script par exemple en javascript. Quand le client reçoit la réponse, le script sera exécuté par le navigateur : il est possible de voler le cookie d'identification ou d'effectuer des actions sur les sites où le client est authentifié. Cette vulnérabilité affecte le client par l'intermédiaire du serveur.

```
Requête :

GET /<SCRIPT>document.write("test");</SCRIPT> HTTP/1.0

Réponse du serveur thttpd :
```

```
HTTP/1.0 404 Not Found
Server: thttpd/2.19 23jun00
Content-type: text/html
Date: Mon, 21 Jun 2004 12:59:24 GMT
Last-modified: Mon, 21 Jun 2004 12:59:24 GMT
Accept-Ranges: bytes
Connection: close

<HTML><HEAD><TITLE>404 Not Found</TITLE></HEAD>
<BODY BGCOLOR="#cc9999"><H2>404 Not Found</H2>
The requested URL '/<SCRIPT>document.write("test");</SCRIPT>'
was not found on this server.
<HR>
<ADDRESS><A HREF="http://www.acme.com/software/thttpd/">
thttpd/2.19 23jun00</A></ADDRESS>
</BODY></HTML>
```

Dans l'exemple ci-dessus, le script utilisé se contente d'afficher la chaîne de caractères « test » dans le navigateur de l'utilisateur.

Dans la suite, nous présentons les vulnérabilités et les exploits affectant notre serveur web dynamique.

4.1.4.2 Attaques et vulnérabilités spécifiques à l'application web

Nous avons introduit quatre vulnérabilités typiques des applications web dans php Bibtex Manager et nous avons développé les exploits correspondants.

La première vulnérabilité consiste en la possibilité d'une injection de code SQL dans la page d'authentification de l'application. Les caractères permettant l'injection de code ne sont pas correctement modifiés avant d'envoyer la requête à la base de données SQL. Il est alors possible d'exécuter n'importe quelle requête SQL que l'application est autorisée à exécuter. Il est notamment possible de s'authentifier auprès de l'application. La requête suivante permet effectivement de s'authentifier en tant qu'administrateur :

```
POST /bibtex/index.php HTTP/1.1
Host: localhost
Connection: close
Content-Length: 43
Content-Type: application/x-www-form-urlencoded

user=admin&pwd=aaaaa") OR (""="&login=Login
```

L'attaquant a ensuite accès à toutes les données gérées par l'application, en lecture et en écriture. Il peut, à loisir, modifier le contenu de la base de données (attaque contre l'intégrité), créer un nouveau compte administrateur, etc.

La deuxième vulnérabilité que nous avons introduite permet l'écriture d'un fichier quelconque à n'importe quel endroit du système de fichiers où l'utilisateur exécutant le processus du serveur HTTP a le droit d'écrire. Il faut d'abord être authentifié auprès de l'application pour exploiter cette vulnérabilité. L'application php Bibtex Manager permet d'envoyer au serveur les fichiers PDF des articles à insérer dans la base. Il est possible de préciser le nom du fichier envoyé. L'application ne vérifie pas, dans ce nom, la présence de caractères permettant une remontée dans l'arborescence (typiquement une suite de ../). Il est ainsi possible d'écrire un fichier quelconque dans un répertoire où l'accès en écriture est autorisé pour l'utilisateur exécutant le processus du serveur HTTP.

```
POST /bibtex/BibtexForm.php HTTP/1.1
Host: 192.168.0.31
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.0.7)
                                        Gecko/20060909 Firefox/1.5.0.7
Accept:text/xml,application/xml,application/xhtml+xml,text/html;
                          q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://192.168.0.31/bibtex/BibtexForm.php
Cookie: PHPSESSID=7b26f5b618bce3782847789621a751f4
Content-Type: multipart/form-data;
    boundary=---------------------------150480625575325511903133490
Content-Length: 2101

...

-----------------------------150480625575325511903133490
Content-Disposition: form-data; name="file"

../../../../../../../../../../../tmp/test3.sh
-----------------------------150480625575325511903133490
Content-Disposition: form-data; name="comments"


-----------------------------150480625575325511903133490
Content-Disposition: form-data; name="fileupload"; filename="test3.sh"
Content-Type: application/octet-stream

#!/bin/bash

echo "Hacked !"
```

```
-----------------------------150480625575325511903133490
Content-Disposition: form-data; name="Insert"

Insert

...
```

La requête précédente (représentée en partie) permet l'écriture d'un fichier dans le répertoire */tmp/*. On peut remarquer aisément la remontée dans l'arborescence.

La troisième vulnérabilité permet l'exécution d'un code quelconque. Il faut au préalable s'être authentifié auprès de l'application. L'exploitation consiste dans le téléchargement d'un fichier php dans un répertoire dans lequel le serveur est configuré pour exécuter des scripts php. C'est le cas par exemple du répertoire où les fichiers PDF sont stockés ou du répertoire parent où sont stockés les scripts php de l'application. Une requête sur la ressource pointant sur le script téléchargé permet alors d'exécuter le code envoyé. Il est possible de créer un shell php et d'exécuter n'importe quelle commande shell, par exemple.

La quatrième vulnérabilité permet de mettre en œuvre une attaque de type Cross-site scripting. L'exploitation de cette vulnérabilité nécessite également d'être authentifié auprès de l'application. Lors de l'ajout d'une entrée dans la base de données des articles, les champs du formulaire ne sont pas filtrés par l'application. Un attaquant peut exploiter ce fait pour mettre dans ces champs des valeurs correspondant à l'exécution de scripts en javascript. Si un autre utilisateur du site affiche l'entrée modifiée par l'attaquant, son navigateur va exécuter le javascript. Il est ainsi possible de voler des cookies ou d'exécuter des requêtes sur les sites où l'utilisateur est connecté. La requête suivante (représentée en partie) montre un exemple où le champ du titre est rempli avec un script récupérant l'heure de l'ordinateur du client et l'affichant.

```
POST /bibtex/BibtexForm.php HTTP/1.1
Host: 127.0.0.1:2000
User-Agent: Mozilla/5.0 (X11;U; Linux i686; en; rv:1.8.1.2)
                          Gecko/20070305 Epiphany/2.16 Firefox/2.0.0.2
Accept: text/xml,application/xml,application/xhtml+xml,text/html;
                                q=0.9,text/plain;q=0.8,image/png,*/*;
                                q=0.5..Accept-Encoding:gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection:keep-alive
Referer: http://127.0.0.1:2000/bibtex/BibtexFormKey.php?key=Jon07c
Cookie: PHPSESSID=5aaac228ebed976ba5c6bc525464f5b5
Content-Type: multipart/form-data;
      boundary=---------------------------92744466811124991479698421 93
Content-Length: 2547
```

```
...

-----------------------------927444668111249914796984219 3
Content-Disposition: form-data; name="title"
   <SCRIPT LANGUAGE="JavaScript">Today=new Date;Heure=Today.getHours();
   Min = Today.getMinutes();Message = "Il est "+Heure+"h"+Min+"m";
   document.write(Message);</SCRIPT>

-----------------------------927444668111249914796984219 3

...
```

Ce type d'attaques affecte au final un utilisateur de l'application mais passe par l'intermédiaire du serveur.

Dans la suite, nous présentons les résultats obtenus par nos deux IDS en utilisant ces sources de trafic « normal », c'est-à-dire sain dans la grande majorité, et ces attaques.

4.2 Résultats de l'IDS de type boîte noire

Quels que soient les tests, nous avons toujours utilisé une architecture composée de trois serveurs. Nous détaillons dans la suite, à chaque fois, les serveurs utilisés et les données utilisées pour rejouer les requêtes. Nous étudions la pertinence et la fiabilité de la détection pour les deux types de serveurs mis en jeu.

4.2.1 Fiabilité de la détection

Nous présentons les tests concernant les 3 vulnérabilités affectant le serveur HTTP minimal et les vulnérabilités affectant Apache, IIS et thttpd. Ensuite, nous présentons les tests sur l'exploitation des vulnérabilités liées au serveur web dynamique. Le trafic HTTP n'était composé que des attaques et celles-ci étaient jouées indépendemment les unes des autres. Ces tests ont été effectués avec nos règles standards de normalisation des requêtes et sans aucune règle de masquage des différences de sortie.

4.2.1.1 Intrusions affectant les serveurs web

Intrusions à l'encontre du serveur HTTP minimal Pour ces tests, l'architecture est composée d'une machine sous Linux sur laquelle fonctionnaient le serveur HTTP minimal, le proxy et l'IDS de type boîte noire. Une autre machine sous Windows 2000 offrait également un service HTTP grâce à un serveur IIS 5.0. Le troisième serveur est un serveur Apache 1.3.29 géré par une machine fonctionnant sous MacOS-X. Les serveurs étaient configurés de manière

homogène : répertoire d'exécution des scripts CGI dans /cgi-bin/, le contenu des répertoires n'est pas listé automatiquement, etc.

La première intrusion est détectée. En effet, le code de réponse est différent pour chaque serveur : le serveur HTTP minimal répond par un code 200 signifiant que la requête a été satisfaite et envoie le contenu du fichier /etc/shadow ; le serveur Apache refuse une requête cherchant à atteindre un document en dehors de l'arborescence servie et répond donc par un code 400 ; le serveur IIS répond, quant à lui, par un code 404 signifiant que la ressource n'a pas été trouvée. La localisation du serveur suivant notre algorithme de comparaison n'est pas possible, tous les codes de statut des réponses étant différents.

La deuxième intrusion est également détectée. L'IDS détecte l'intrusion et est, de plus, capable d'indiquer quel serveur a été attaqué puisque les serveurs Apache et IIS refusent la requête la considérant comme mal-formée et renvoie un code 400 pour le signaler au client et le serveur HTTP minimal l'accepte en renvoyant un code 200.

Pour la troisième intrusion, nous avons modifié le serveur HTTP minimal comme indiqué précédemment et effectué un débordement de tampon dans le but de faire défaillir le serveur. Cette intrusion est détectée par le chien de garde car le serveur HTTP minimal ne répond pas à la requête. La localisation du serveur échoue car le serveur Apache répond par un code de statut 400 pour indiquer que la requête est mal-formée alors que le serveur IIS répond par un code de statut 414 pour indiquer que l'URL est trop longue.

L'intrusion à l'encontre du serveur Apache Pour ce test, l'architecture était composée d'un serveur Linux sur lequel fonctionnaient le proxy, l'IDS, un serveur Apache 1.3.12 (vulnérable à l'attaque) et un serveur thttpd 2.25b. Un serveur IIS 5.0 fonctionnant sur une machine sous Windows 2000 constitue la troisième variante. Dans notre cas, il faut envoyer 4066 '/' pour obtenir la liste des répertoires.

L'intrusion est détectée mais la localisation du serveur n'est pas possible. En effet, le serveur thttpd considère la requête comme mal-formée et répond par un code de statut 400. Le serveur IIS, quant à lui, répond en envoyant la page d'accueil du site (et donc un code de statut 200). Le serveur Apache répond par un code 200 mais envoie le contenu du répertoire racine du serveur web.

Intrusions à l'encontre du serveur IIS Nous avons exploité la vulnérabilité affectant IIS de deux manières, pour attaquer d'une part la confidentialité du serveur et d'autre part son intégrité. Dans la première attaque, nous accédons au contenu du répertoire `C:` du serveur Windows et, dans la seconde, nous copions le fichier cmd.exe vers un fichier cmd1.exe. Pour ce test, l'architecture est composée d'un serveur Linux sur lequel fonctionnent le proxy, l'IDS et un serveur thttpd 2.25b. La seconde variante était constituée d'un serveur Apache 1.3.29 sous MacOS-X. La troisième variante vulnérable était constituée d'un serveur IIS 5.0 sous Windows 2000.

Les deux intrusions sont détectées : le serveur IIS répond par un code 500

et envoie le résultat des deux commandes exécutées pour obtenir le contenu du répertoire ou pour copier le fichier. Les deux autres serveurs répondent par des codes d'erreurs 404. L'IDS est de plus capable de localiser le serveur compromis pour ces deux intrusions.

L'intrusion à l'encontre du serveur thttpd L'exploitation de cette vulnérabilité ne cherche pas à affecter le serveur mais un client potentiel du site, grâce à un vol de cookie par exemple. Le serveur peut ensuite être affecté si l'attaquant réussit à récupérer la session d'un utilisateur normal. Cette attaque s'applique notamment à la version 2.19 du serveur. L'architecture, pour ce test, se composait d'une machine sous Linux gérant le proxy, l'IDS et le serveur thttpd 2.19, une machine sous Windows 2000 gérant un serveur IIS 5.0 et une machine sous MacOS-X gérant un serveur Apache 1.3.29.

L'intrusion n'est pas détectée : tous les serveurs répondent par un code de statut 404. Comme nous avons décidé de ne pas comparer le contenu envoyé dans ce cas-là, l'IDS n'est pas capable de voir que le serveur thttpd ne gère pas correctement les caractères spéciaux alors que c'est le cas pour le serveur Apache. Si on utilise seulement des serveurs pour lesquels il est possible de définir chaque contenu correspondant à des codes d'erreurs, il est alors possible de comparer le contenu même en cas de codes d'erreurs et éventuellement de détecter ce genre d'attaques.

4.2.1.2 Intrusions affectant l'application web

Nous avons testé les quatre vulnérabilités introduites dans php Bibtex Manager. L'architecture était composée d'un serveur Lighttpd 1.4.13 sous Linux, d'un Apache 1.3.31 sous MacOS-X, d'un serveur Abyss 2.0.6 sous Windows 2000. Nous avons introduit chaque vulnérabilité dans une des versions de php Bibtex Manager.

La première vulnérabilité a été introduite sur le serveur Abyss. L'injection de code SQL est détectée : tous les serveurs envoient un code 302 pour rediriger le client vers une page spécifiée dans l'en-tête *Location*. Les serveurs Apache et Lighttpd renvoient vers la page de login alors que le serveur Abyss renvoie vers la page principale de l'application. Le serveur Abyss envoie également un cookie au client par l'intermédiaire de l'en-tête *Set-Cookie*. L'IDS détecte l'intrusion et est capable de localiser le serveur compromis.

La deuxième vulnérabilité a été introduite sur le serveur Lighttpd. Les versions non-vulnérables enlèvent les séquences de « ../ » permettant la remontée dans l'arborescence. Les trois serveurs acceptent la requête et renvoient une page contenant le chemin du fichier envoyé. Ce chemin est différent pour le serveur Lighttpd qui n'a pas filtré les caractères ../ correctement. Nous avons vérifié que dans le cas où la réponse du serveur ne contient que le message « *file uploaded* », l'intrusion n'est pas détectée.

La troisième vulnérabilité a été introduite sur le serveur Abyss. Les versions non-vulnérables ajoutent l'extension « .txt » au fichier envoyé si celui-ci n'a pas une extension autorisée « .pdf », « .ps », etc. La version vulnérable ne fait pas

	serveur HTTP minimal 1	serveur HTTP minimal 2	serveur HTTP minimal 3	Apache	IIS 1	IIS 2	thttpd (XSS)
Alerte	Oui	Oui	Oui	Oui	Oui	Oui	Non

TABLE 4.1 – Résultats de détection de l'IDS boîte noire pour les intrusions affectant les serveurs

	injection SQL	écriture	exécution d'un script php	XSS (insertion)	XSS (lecture de l'entrée corrompue)
Alerte	Oui	Oui	Oui	Non	Oui

TABLE 4.2 – Résultats de détection de l'IDS boîte noire pour les intrusions affectant l'application web

cet ajout. Nous avons modifié php Bibtex Manager pour ne pas renvoyer le nom du fichier en réponse à l'envoi du fichier, ainsi l'intrusion n'est pas détectée à ce moment là. L'attaquant fait ensuite une requête sur le fichier qu'il a téléchargé. Celui-ci n'est pas trouvé par le serveur thttpd. Il est trouvé par le serveur Abyss qui exécute le code php téléchargé. Le serveur Apache présente ici une différence de spécification par rapport à thttpd par exemple : le fichier demandé n'existe pas sur son disque dur car l'extension « .txt » a été ajouté mais Apache renvoie quand même le fichier sous forme de texte. Les trois réponses des serveurs sont différentes : l'intrusion est donc détectée. La localisation est, cependant, impossible.

La quatrième vulnérabilité a été introduite dans Apache. Lors de l'insertion dans la base SQL d'une entrée contenant du code javascript, les serveurs répondent tous de la même manière et l'intrusion n'est donc pas détectée. Par contre, lors de la lecture de l'entrée modifiée, une différence dans le contenu des réponses est détectée car les serveurs non-vulnérables ont modifié l'entrée pour qu'elle ne corresponde plus à du code javascript exécutable par le navigateur de l'utilisateur.

Les tableaux 4.1 et 4.2 résument les résultats de détection. La majorité des intrusions sont détectées car elles ont un effet sur les réponses réseau des différents serveurs. Ils mettent aussi en évidence les points faibles de notre approche boîte noire. L'absence de comparaison des contenus dans tous les cas conduits au premier faux négatif noté thttpd (XSS) dans le premier tableau. Le second faux négatif est lié à une intrusion qui affecte l'intégrité d'un des serveurs : le serveur compromis a une entrée de sa base de données qui est corrompue. Cette intrusion n'entraîne pas de réponses immédiatement différentes de la part des

serveurs. Par contre, quand cette entrée est envoyée à un autre client (ce qui peut être bien après l'insertion de l'entrée), l'IDS boîte noire est capable de détecter l'intrusion. Un diagnostic est alors nécessaire pour comprendre l'alerte. Le client qui demande la page n'est pas l'attaquant mais certainement une victime redirigée sur cette page par l'attaquant.

Ces tests montrent ainsi que notre approche de type boîte noire peut effectivement détecter des intrusions. Nous étudions dans la suite la pertinence de la détection liée au taux de faux positifs.

4.2.2 Pertinence de la détection

Pour tester la pertinence de la détection, nous avons testé notre IDS de type boîte noire avec plusieurs jeux de données correspondant aux serveurs web statiques et dynamiques. Dans le cadre de nos tests sur le serveur web statique, nous avons utilisé les logs des requêtes datant de mars 2003 et les logs des requêtes datant de 2006. Dans le cadre de nos tests sur le serveur web dynamique, nous avons utilisé le trafic obtenu « artificiellement » décrit précédemment.

4.2.2.1 Résultats pour le serveur web statique

Logs de mars 2003 Pour ce test, l'architecture était composée d'un serveur Apache 1.3.29 sous MacOS-X, d'un serveur thttpd 2.25b sous Linux et un serveur IIS 5.0 sous Windows 2000. Le proxy et l'IDS fonctionnaient sur la machine sous Linux. Les trois serveurs contenaient une copie du contenu du serveur web datant de mars 2005 de Supélec Rennes (voir 4.1.1) et ont été configurés de manière la plus homogène possible de sorte à générer un minimum de faux positifs : les serveurs ne génèrent pas le contenu des répertoires dynamiquement par exemple. Le test était constitué des requêtes provenant des logs du serveur web du campus de Rennes de Supélec datant de mars 2003, ce qui représente un peu plus de 800 000 requêtes. Une étude précédente [TDMD04] portant sur les mêmes logs et utilisant un outil de détection d'intrusions très sensible montre qu'au plus 1,4% des requêtes peuvent être considérées comme dangereuses.

Comme le montre la figure 4.1, 99,67% des différences entre les réponses des serveurs sont masquées. Seules 36 règles sont utilisées pour le masquage des différences de sortie. 5 règles seulement représentent près de 90% des différences masquées. Il ne faut donc que peu de règles pour rendre le système viable, même s'il est probable que toutes les différences de spécification n'aient pas été rencontrées et donc n'aient pas été masquées. Sans l'utilisation de règles, les différences entre les réponses des serveurs auraient causé un grand nombre de faux positifs. Cela met en évidence les difficultés liées à l'utilisation de COTS dans un système N-versions pour la détection.

Seules 0,37% des différences entre les réponses des serveurs génèrent une alerte ; cela représente 0,016% des requêtes HTTP effectuées et correspond à peu près à 5 alertes par jour. Le responsable de la sécurité doit ensuite analyser ces alertes pour en déterminer les causes et éliminer les faux positifs potentiels

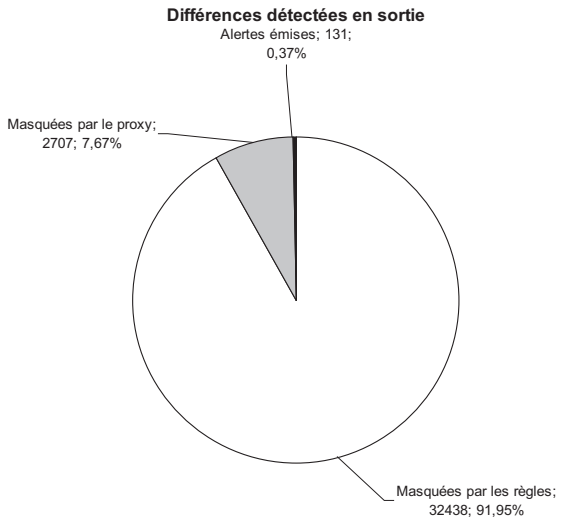


FIGURE 4.1 – Analyse des différences détectées. Sur les 800 000 requêtes constituant le test, 35000 requêtes génèrent des différences détectées par l'IDS sans mécanisme de masquage. Avec les règles de masquage, seules 131 alertes sont levées.

en ajoutant d'éventuelles règles de masquage. Parmi les alertes obtenues lors de notre expérimentation (voir FIG. 4.2), certaines sont, en effet, probablement des faux positifs : les quatre derniers types d'alertes correspondent en effet certainement à des attaques qui n'ont pas réussi et donc n'auraient pas du engendrer d'alertes puisque l'IDS que nous proposons ne doit lever une alerte qu'en cas d'intrusion effective sur l'un des serveurs. Cela représente 22% des alertes. Ces attaques sont composées de tentatives d'utiliser les fonctionnalités de proxy de certains serveurs web (comme mod_proxy pour Apache) : ce sont les requêtes du type '`GET http://url/`'. Les requêtes de type '`CONNECT site`' sont également des tentatives d'utilisation de certaines fonctionnalités de proxy que peuvent jouer les serveurs web. Les « requêtes encodées » correspondent à des requêtes dont l'URL a été encodée dans le but de déjouer les mécanismes de protection des serveurs web et ainsi accéder à des fichiers non servis par le serveur web normalement. Les quatre premiers types d'alertes correspondent bien à des intrusions contre le serveur IIS : ce sont des attaques contre la confidentialité du serveur vraisemblablement générées par des scripts vérifiant la vulnérabilité du serveur ou non.

Sur cet ensemble de requêtes, nous avons comparé les résultats de notre approche à ceux de deux autres IDS connus, WebSTAT [VRKK03] et Snort [Roe99], configurés avec leur ensemble de signatures par défaut. Cette comparaison reste

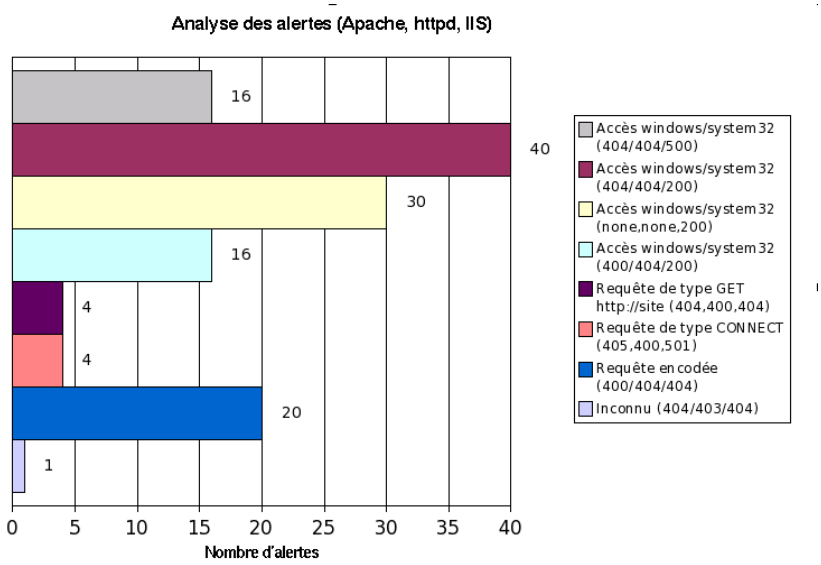


FIGURE 4.2 – Analyse des alertes

donc limitée car il est nécessaire de configurer la base de signature des IDS suivant cette approche. Il n'est pas facile de comparer des IDS : ils ne possèdent pas le même ensemble de signatures (WebSTAT et Snort) et produisent des faux positifs différents. La figure 4.3 montre les alertes émises par jour par chaque IDS. En moyenne, WebSTAT et Snort émettent 10 alertes par jour alors que notre IDS n'en émet que 5. Ceci peut s'expliquer par le fait que Snort et WebSTAT émettent des alertes lors d'attaques, c'est-à-dire de tentatives d'intrusion qui peuvent réussir ou non. Notre IDS est censé ne détecter que des intrusions, c'est-à-dire des fautes provenant de l'exploitation d'une vulnérabilité et qui entraînent une violation de la politique de sécurité ; toutefois certaines alertes sont des faux positifs qui correspondent effectivement à des attaques ayant échoué qui sont aussi détectées par Snort ou WebSTAT.

L'analyse des alertes montre qu'une partie des alertes sont communes aux 3 IDS et correspondent bien à des intrusions contre le serveur IIS. Les autres alertes sont distinctes et celles émises par WebSTAT et Snort sont soit de « véritables » faux positifs soit des alertes correspondant bien à des attaques mais qui ne causent pas d'intrusions dans l'un des serveurs. Les IDS par signature, sans mécanismes complémentaires comme par exemple celui décrit dans [KRV04], ne peuvent généralement faire la différence entre alertes et intrusions. Cette analyse laisse à penser que notre IDS de type boîte noire n'a pas manqué d'intrusions sur cet ensemble. Si ce n'est pas le cas, les trois IDS ont manqué l'intrusion.

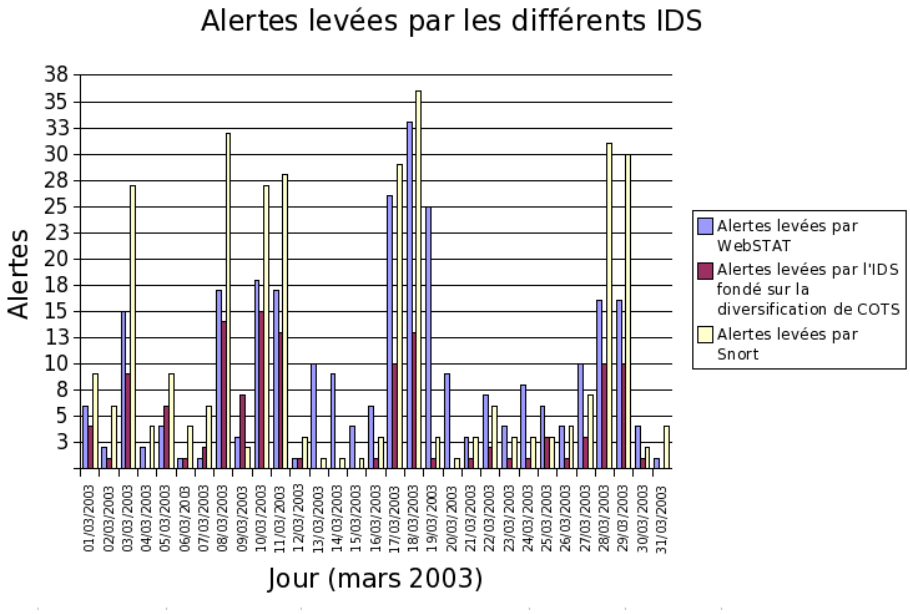


FIGURE 4.3 – Comparaison des résultats de l'IDS de type boîte noire avec Snort et WebSTAT sur les logs datant de mars 2003

Logs de septembre-octobre 2006 En utilisant les mêmes serveurs pour l'architecture et le même contenu pour les serveurs web, nous avons mené d'autres tests en utilisant les requêtes présentes dans les logs de septembre-octobre 2006. Nous avons utilisé les deux semaines constituant ce jeu de données. La première semaine a été utilisée pour l'apprentissage des règles et la seconde pour tester l'IDS et les règles mises en place. Comme expliqué précédemment, l'apprentissage des règles n'est pas automatique : l'administrateur écrit les règles qui conviennent. Il n'est donc pas nécessaire de vérifier que la semaine d'apprentissage soit exempte d'intrusions. La semaine d'apprentissage comprend 71 596 requêtes HTTP. La semaine utilisée pour le test comprend 105 228 requêtes. Lors de cette semaine de test, l'IDS de type boîte noire a levé 50 alertes, ce qui représente donc 7 alertes par jour. Après analyse, toutes ces alertes se sont révélées être des faux positifs.

Les règles de masquage au niveau de l'IDS prouvent leur nécessité et leur efficacité (voir figure 4.4). Sans règles de masquage de différences de sortie, 5975 alertes auraient été levées (ce qui aurait représenté 853 alertes par jour). Ceci signifie que 99,16% des différences de sortie ont été masquées par des règles.

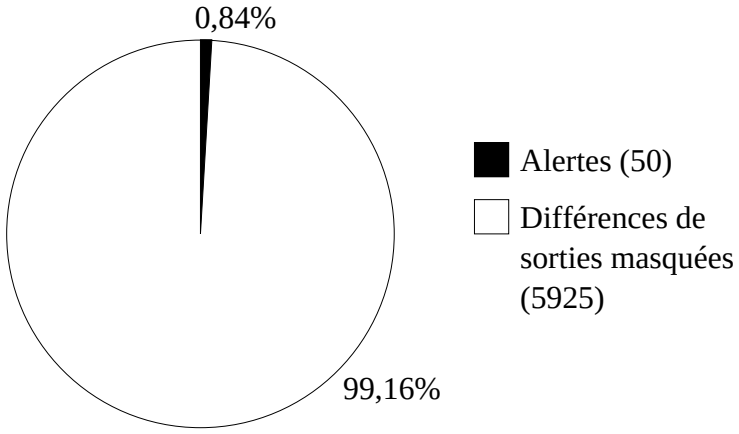


FIGURE 4.4 – Analyse des différences détectées. Sur les 800 000 requêtes constituant le test, 35000 requêtes génèrent des différences détectées par l'IDS sans mécanisme de masquage. Avec les règles de masquage, seules 131 alertes sont levées.

4.2.2.2 Résultat pour le serveur web dynamique

L'architecture était composée d'un serveur Lighttpd 1.4.13 sous Linux, d'un Apache 1.3.31 sous MacOS-X, d'un serveur Abyss 2.0.6 sous Windows 2000. Les règles de masquage des différences en sortie utilisées sont celles déterminées pour le jeu de trafic réel utilisé précédemment. Dans ce test, on utilise le trafic que nous avons sauvegardé, destiné à l'application php Bibtex Manager (voir 4.1.3). Comme expliqué en 3.2.2, nous avons du modifier php Bibtex Manager pour que le champ date ne soit plus présent dans les réponses.

Dans cette configuration, notre IDS de type boîte noire ne lève aucune alerte pour les 89 requêtes saines présentes dans ce jeu de données.

Les résultats de notre IDS de type boîte noire sont relativement bons que ce soit du point de vue de la pertinence de détection ou de la fiabilité de détection. En effet, très peu d'intrusions ne sont pas détectées par l'IDS dans nos tests de fiabilité. De plus, le taux de faux positifs semble raisonnable. Si l'administrateur établit des règles de masquage au fur et à mesure, cela contribue à diminuer le taux de faux positifs. Nous étudions dans la suite les résultats de notre approche boîte grise.

4.3 Résultats de l'IDS de type boîte grise

Notre IDS de type boîte grise a été développée après notre IDS de type boîte noire : moins de tests ont pu être menés sur cet IDS. Cet IDS nécessite, de plus, une phase d'apprentissage qui sert à déterminer les différents seuils de

Similarité entre \ Intrusions	injection SQL	écriture	exécution d'un script php	XSS (insertion)	XSS (lecture de l'entrée corrompue)
Lighttpd et Apache	0.9655	0.725	0.5882	0.9677	0.9143
Apache et Abyss	0.7742	0.9715	0.6364	0.9677	0.9706
Lighttpd et Abyss	0.7838	0.7209	0.6667	1	0.9189

TABLE 4.3 – Similarités entre les graphes des différents serveurs dans le cas des intrusions

similarité. Il n'est pas possible de tester la fiabilité seule comme nous l'avons fait pour l'IDS de type boîte noire (il suffirait de placer le seuil de similarité juste au-dessus de la plus faible des similarités calculées entre les graphes pour détecter les attaques). Ainsi il est nécessaire pour cet IDS de tester la fiabilité en fonction de la pertinence. C'est ce que nous faisons tout d'abord en étudiant sur le trafic « artificiel » sauvegardé pour l'application web php Bibtex Manager. Par la suite, nous testons la pertinence de cet IDS sur un jeu de données.

4.3.1 Fiabilité de la détection

Nous testons la fiabilité et la pertinence de notre IDS de type boîte grise sur l'application web php Bibtex Manager. Nous calculons la moyenne des similarités pour les requêtes normales que nous avons sauvegardées et la similarité pour les quatre intrusions que nous avons développées.

Nous avons testé les quatre vulnérabilités introduites dans php Bibtex Manager. L'architecture était composée d'un serveur Lighttpd 1.4.13 sous Linux, d'un Apache 1.3.31 sous MacOS-X, d'un serveur Abyss 2.0.6 sous Windows 2000. Le proxy et l'IDS fonctionnait sur la machine sous Linux. Nous avons introduit chaque vulnérabilité dans une des versions de php Bibtex Manager de la même manière qu'en 4.2.2.2.

Les similarités entre les différents graphes correspondant aux intrusions (à l'exception du XSS) sont inférieures à celles des requêtes normales. En moyenne, pour les requêtes normales, la similarité est environ de 0,95, alors que, pour les intrusions, celle-ci est inférieure à 0,8. L'IDS boîte grise que nous proposons est incapable, comme l'IDS boîte noire, de détecter le XSS. Cependant, contrairement à l'IDS boîte noire, il ne détecte pas non plus cette intrusion lorsqu'une requête demande une page contenant l'entrée corrompue. En effet, dans ce cas, le comportement des serveurs est proche en terme de flux d'informations : seules les données envoyées diffèrent. Le tableau 4.3 résume les différentes similarités obtenues pour les intrusions.

La localisation du serveur compromis est immédiatement visible à partir des similarités dans les cas des deux premières intrusions. Dans le cas de l'injection SQL, les similarités concernant le serveur Abyss sont plus faibles et dans le cas de l'écriture d'un fichier par remontée dans l'arborescence, les similarités

concernant le serveur Lighttpd sont plus faibles. Ce sont bien les deux serveurs affectés par les intrusions. Ce n'est pas le cas dans la troisième intrusion testée à cause de différences de spécifications entre les serveurs qui entraînent une similarité basse entre tous les serveurs. La localisation n'est donc pas possible dans ce cas : tous les serveurs sont considérés comme compromis.

En plaçant le seuil pour chaque couple de serveurs à 0,8, notre prototype est capable de détecter toutes les intrusions à l'exception du XSS et lève 6 fausses alertes lorsque l'on rejoue les 86 requêtes normales enregistrées. En fixant les seuils à 0,85, toutes les intrusions à l'exception du XSS sont détectées et l'IDS lève 9 fausses alertes.

Ces tests montrent que notre prototype est capable de détecter des intrusions, lorsque celles-ci impliquent une différence dans le comportement des serveurs. Actuellement, le taux de faux positifs reste sensiblement élevé dans ce test.

4.3.2 Pertinence de la détection

Dans ce test, nous ne testons que la pertinence de l'IDS de type boîte grise dans le cadre du serveur web statique en utilisant un jeu de données réelles.

4.3.2.1 Logs de septembre-octobre 2006

Pour ce test, l'architecture était composée d'un serveur Apache 1.3.29 sous MacOS-X, d'un serveur thttpd 2.25b sous Linux et un serveur IIS 5.0 sous Windows 2000. Le proxy et l'IDS fonctionnaient sur la machine sous Linux. Les trois serveurs contenaient une copie du contenu du serveur web de Supélec Rennes datant de mars 2005. Nous avons procédé de la même manière que pour l'IDS de type boîte noire pour ce jeu de données : nous avons utilisé la première semaine qui contient 71 596 requêtes comme semaine d'apprentissage et la seconde semaine qui contient 105 228 requêtes comme semaine de test. La semaine d'apprentissage est nécessaire pour déterminer les seuils de similarité pour chaque couple de serveurs.

Détermination des seuils $t_{i,j}$ Comme expliqué en 2.3.4.2, les seuils $t_{i,j}$ doivent être choisis expérimentalement pour chaque couple de serveurs utilisés. Nous avons décidé de les fixer de telle manière que, pour la plupart des requêtes HTTP, notre prototype ne lève pas d'alertes, c'est-à-dire que la majorité des similarités calculées soit supérieure aux seuils $t_{i,j}$.

Contrairement à notre IDS de type boîte noire, il est nécessaire de vérifier que l'ensemble des requêtes constituant la semaine d'apprentissage est dénué d'un grand nombre d'intrusions sur nos serveurs. Pour vérifier que cet ensemble ne contient pas d'intrusions contre l'un des serveurs utilisés, nous avons utilisé WebSTAT [VRKK03]. WebSTAT lève 33 alertes qui se sont révélées être des faux positifs. Cette vérification n'est pas sûre mais c'est la seule méthode possible puisqu'il n'est pas envisageable d'analyser toutes les requêtes manuellement.

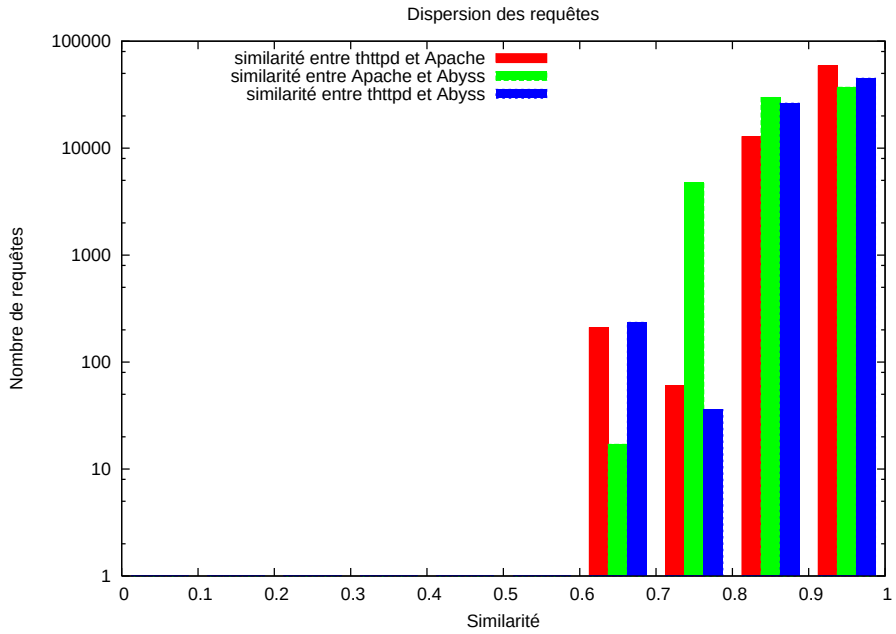


FIGURE 4.5 – Répartition des requêtes en fonction de la similarité pour chaque couple de serveurs (le nombre de requêtes est exprimé en échelle logarithmique)

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 4.4 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 L'ensemble constituant la semaine de test 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 architecture et la dernière ne correspond pas à une attaque et est donc un faux positif. Le tableau 4.5 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% soit 20 alertes par jour.

Ces résultats montrent les capacités de détection de notre IDS de type boîte

	$s_{2,3} \in I_1^{2,3}$		$s_{2,3} \in I_2^{2,3}$	
$s_{1,2} \in$ \ $s_{1,3} \in$	$I_1^{1,3}$	$I_2^{1,3}$	$I_1^{1,3}$	$I_2^{1,3}$
$I_1^{1,2}$	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 4.4 – 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

	$s_{2,3} \in I_1^{2,3}$		$s_{2,3} \in I_2^{2,3}$	
$s_{1,2} \in$ \ $s_{1,3} \in$	$I_1^{1,3}$	$I_2^{1,3}$	$I_1^{1,3}$	$I_2^{1,3}$
$I_1^{1,2}$	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 4.5 – 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

grise. Les intrusions contre l'application web (excepté le XSS) sont détectées sans que le taux de faux positifs ne soit trop élevé. Pour notre test de pertinence, le taux de faux positif est acceptable bien que cela représente 20 alertes par jour à traiter par l'administrateur. Un mécanisme de masquage des différences de spécification/conception peut améliorer les résultats de pertinence de détection. Dans la suite, nous comparons les résultats des deux approches sur les tests communs.

4.4 Comparaison des deux approches proposées

Il n'a pas été possible de tester sur l'ensemble des mêmes jeux de données les deux IDS pour deux raisons. D'une part, il ne nous a pas été possible d'instrumenter le serveur web IIS pour construire les graphes de flux d'informations. Il ne nous a pas été ainsi possible de refaire avec notre prototype d'IDS de type boîte grise le premier de test pertinence effectué sur notre approche boîte noire (voir 4.2.2). D'autre part, tester la fiabilité de notre IDS de type boîte grise n'a pas de sens sans tester à la fois sa pertinence car la détection dépend d'un seuil. Les tests de fiabilité effectués sur notre IDS de type boîte noire pour les intrusions affectant les serveurs web (voir 4.2.1.1) n'ont pu être menés. En effet, il n'est pas raisonnable de rejouer de véritables requêtes sur le serveur HTTP minimal car celui-ci ne répond qu'aux requêtes de type GET. Les autres intrusions

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

TABLE 4.6 – Corrélation entre les deux approches sur la semaine de test

affectaient soit le serveur IIS qu'il ne nous a pas été possible d'instrumenter soit des anciennes versions de serveurs Apache ou thttpd.

4.4.1 Comparaison et Corrélation

Nous avons mené deux tests communs aux deux IDS : notre test sur le jeu de données de septembre-octobre 2006 et notre test sur le serveur web dynamique faisant tourner l'application web php Bibtex Manager.

Logs de septembre-octobre 2006 Nous étudions tout d'abord les résultats sur les logs de septembre-octobre 2006. Les résultats individuels sont donnés en 4.2.2 et en 4.3.2.1. Pour la seconde semaine qui constitue la semaine de test, l'IDS boîte noire lève 50 alertes et l'IDS boîte grise 140 alertes (voir tableau 4.6). Il est nécessaire de rappeler qu'aucune de ces requêtes n'est une intrusion. La combinaison des deux IDS lève 180 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.

Résultat pour le serveur web dynamique Nous avons également testé les deux approches sur le serveur web dynamique. Les résultats individuels sont décrits en 4.2.1.2 et en 4.2.2.2 pour l'IDS de type boîte noire et en 4.3.1 pour l'IDS de type boîte grise. Nous avons considéré le cas où les seuils de similarité étaient fixés à 0,8.

Nous avons regroupé les résultats pour les 86 requêtes normales et cinq requêtes constituées par les quatre intrusions en considérant chaque requête jouée une fois. Le tableau 4.7 regroupe les résultats des deux IDS. Les trois alertes communes sont pour les trois premières intrusions. Les six alertes levées seulement par l'IDS de type boîte grise sont les six faux positifs de l'IDS dans ce test. Seule l'intrusion XSS n'est détectée par aucun des deux IDS et complète les 80 requêtes normales et considérées comme normales par les deux IDS.

4.4.2 Discussion

Seuls deux tests ont pu être menés sur les deux approches à la fois. Il semble nécessaire de réaliser des tests plus nombreux pour pouvoir évaluer les capacités de corrélation des deux approches.

IDS boîte noire \ IDS boîte grise	alerte	pas d'alerte
alerte	3 (A)	1 (A)
pas d'alerte	6 (A)	81 (NA)

TABLE 4.7 – Corrélation entre les deux approches pour le serveur web dynamique

L'IDS de type boîte grise est censé avoir une plus grande couverture de détection du fait d'observations plus précises que les seules sorties des serveurs COTS. Les tests effectués ne permettent pas de conclure puisque l'IDS de type boîte grise tout comme l'IDS de type boîte noire ne sont pas capables de détecter l'intrusion XSS lorsque se produit l'insertion du javascript dans la base de données. L'IDS de type boîte noire est même capable de détecter l'intrusion XSS lorsque la donnée corrompue est lue et envoyée au client alors que ce n'est pas le cas de l'IDS de type boîte grise. Il faut cependant nuancer ces propos car la prise en compte des données des flux d'informations peut permettre de détecter ces deux intrusions alors que l'IDS de type boîte noire n'est pas en mesure de détecter l'insertion de la donnée corrompue dans la base de données.

L'IDS de type boîte grise lève plus de faux positifs que l'IDS de type boîte noire. Cela semble logique dans le sens où il est plus sensible que l'IDS de type boîte noire et que nous n'avons pas développé de mécanismes de masquage des différences comme nous l'avons fait pour l'IDS de type boîte noire. Ainsi, les différences de spécification/conception qui entraînent une différence importante entre les graphes de flux d'informations des différents serveurs va forcément entraîner une alerte. Notre mécanisme de masquage des différences de sortie peut éventuellement empêcher une alerte d'être levée dans ce cas.

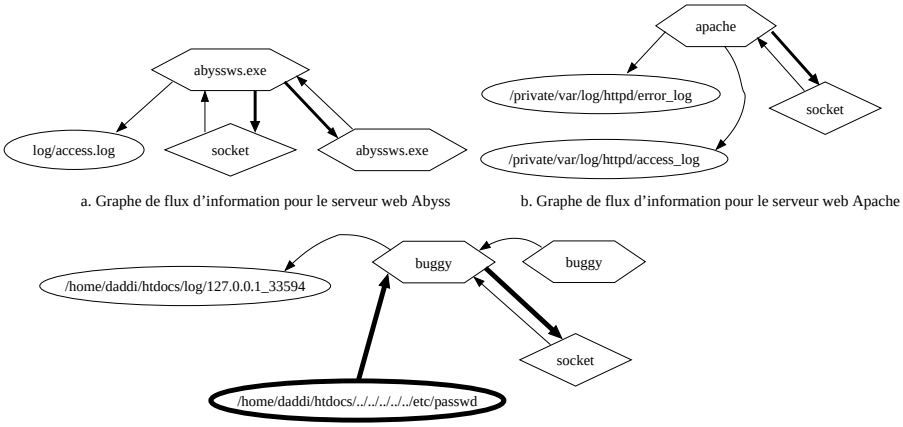
L'autre avantage de notre IDS de type boîte grise est sa capacité de diagnostic : nous donnons quelques exemples de diagnostic dans la suite.

4.5 Aide au diagnostic

Dans cette section, nous évaluons sur quelques exemples d'intrusions les capacités d'aide au diagnostic qu'apporte notre approche de type boîte grise. Comme expliqué en 2.3.5, l'aide au diagnostic consiste en la mise en évidence des objets et flux non associés dans le meilleur mapping.

La figure 4.6 montre un exemple simple d'une intrusion contre la confidentialité d'un serveur web vulnérable qui a été développé dans un but pédagogique. L'intrusion consiste en une attaque par remontée dans l'arborescence pour aller lire le fichier */etc/passwd*. Le serveur vulnérable ne vérifie pas la présence de '../' dans la chaîne de caractères de l'URL alors que les deux autres serveurs refusent de traiter une telle requête et renvoient une erreur au client.

Les meilleurs mappings pour chaque couple de serveurs sont donnés dans les figures 4.7, 4.8 et 4.9.



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 HTTP minimal

FIGURE 4.6 – Trois graphes de flux d'informations liés à une même requête HTTP traitée sur différents serveurs et l'identification des objets et des flux qui ne sont pas associés

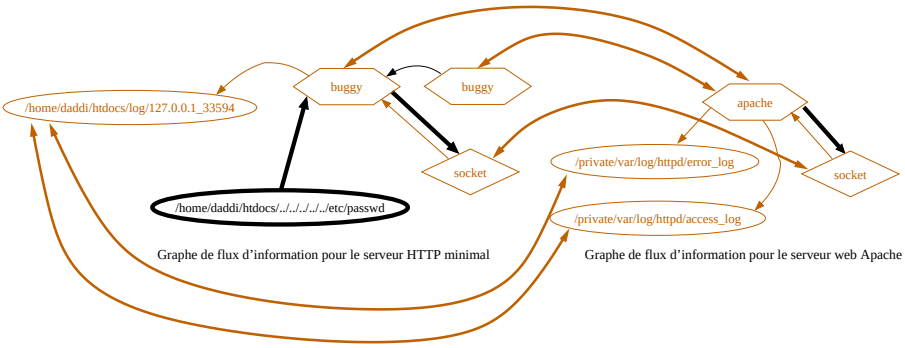


FIGURE 4.7 – Meilleur mapping entre les graphes du serveur Apache et du serveur HTTP minimal

L'objet fichier représentant le fichier */etc/passwd* n'est associé avec aucun des objets dans les autres graphes. C'est le cas également pour le flux d'information représentant l'accès en lecture au fichier */etc/passwd* et l'écriture sur la socket. Ces objets sont mis en évidence sur la figure 4.6 (trait très gras). Pour le serveur Apache comme pour le serveur Abyss, les flux d'informations représentant l'écriture sur la socket ne sont pas associés avec leur équivalent dans le graphe du serveur vulnérable mais sont associés ensemble dans le mapping concernant Abyss et Apache. Ils sont donc mis en valeur mais d'une manière moins visible (trait gras). Dans cet exemple, les objets et flux mis en valeur sont

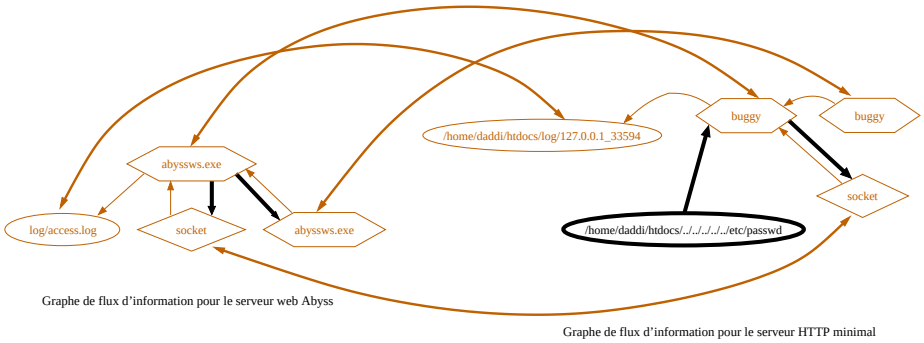


FIGURE 4.8 – Meilleur mapping entre les graphes du serveur Abyss et du serveur HTTP minimal

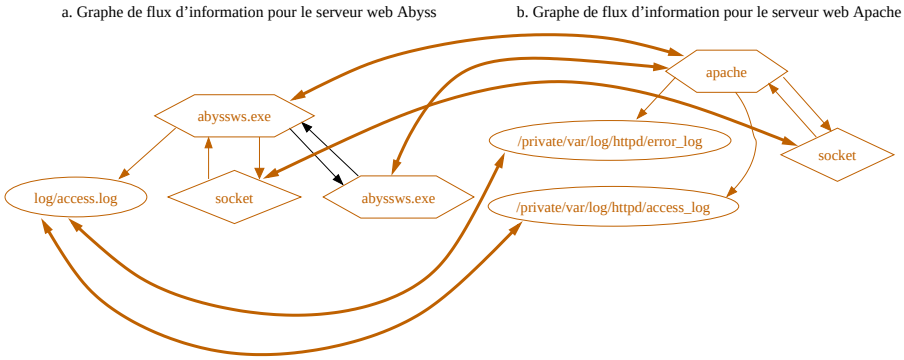


FIGURE 4.9 – Meilleur mapping entre les graphes des serveurs Apache et Abyss

bien les effets de l'intrusion au niveau du système d'exploitation.

La figure 4.10 montre un autre exemple d'aide au diagnostic sur l'injection de code SQL dans l'application php Bibtex Manager. Cette injection de code permet à l'utilisateur de s'authentifier auprès de l'application web en tant qu'administrateur. Les manifestations de l'intrusion au niveau des flux d'informations au niveau du système d'exploitation semblent moins importants que sur l'exemple précédent. Les seuls objets et flux non associés dans les trois meilleurs mappings sont le flux d'écriture sur la socket client et le fichier de création de session et l'écriture d'informations dans ce fichier. Cette écriture est le symptôme de l'intrusion au niveau des flux d'informations. Elle marque le fait que le serveur compromis crée une session pour pour le client et écrit les paramètres de la session dans le fichier */tmp/sess_0015b077bca48135036c579b575179d4*. Si les données circulant lues et écrites par les processus étaient prises en compte, les flux d'informations modélisant la requête à la base SQL et la réponse de cette base n'auraient pas été associés dans le cas du serveur compromis. Il est

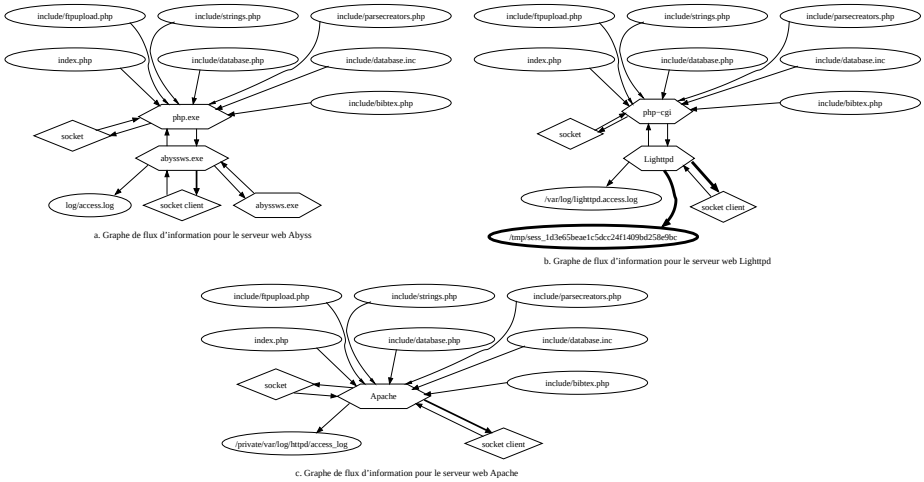


FIGURE 4.10 – Trois graphes de flux d'informations liés à l'injection SQL contre php Bibtex Manager et l'identification des objets et des flux qui ne sont pas associés

moins évident en regardant les graphes et la mise en évidence des objets et flux non associés dans les meilleurs mappings de trouver la nature de l'intrusion. L'approche indique tout de même à l'administrateur de vérifier les réponses des serveurs et le contenu de ce fichier, ce qui donne des premières pistes à l'administrateur de sécurité pour commencer son analyse de l'alerte.

Nous pensons que cette approche peut être réellement bénéfique à un administrateur de sécurité et peut être étendue pour automatiquement établir un résumé des interactions impliquant les objets non associés.

4.6 Performances

Nous évaluons le coût en performances introduit par l'architecture : nous avons donné un modèle théorique permettant d'évaluer ce coût en 2.1.6. Ici, nous présentons les résultats des prototypes que nous avons développés. Il faut noter que ceux-ci ne sont pas optimisés dans leur conception et par le langage utilisé, *perl*, qui a été utilisé pour sa facilité de programmation. Nous présentons les performances de l'IDS de type boîte noire puis de l'IDS de type boîte grise.

4.6.1 Performances de l'IDS de type boîte noire

Pour mesurer les performances de l'IDS de type boîte noire, nous avons mesuré du point de vue du client le coût en temps de réponse de l'architecture de détection d'intrusions et de tolérance aux intrusions que nous proposons. Pour cela, nous avons utilisé le jeu de données de mars 2003 en utilisant trois

serveurs : un thttpd 2.25b sur Linux, un Apache 1.3.29 sous MacOS-X, un IIS 5.0 sous Windows 2000. Le proxy et l'IDS fonctionnaient sur la machine Linux, ce qui peut induire un temps de latence supplémentaire si le processeur n'est pas assez puissant pour gérer à la fois la comparaison, le proxy et le serveur web. Les trois serveurs contenaient une copie du contenu du serveur web datant de mars 2005 de Supélec Rennes.

Le tableau 4.8 résume ces mesures. Les mesures donnent le temps de réponse de l'architecture : le temps entre l'envoi de la requête du client et la réception de la réponse. La ligne « Sans l'IDS » du tableau donne la moyenne des temps de réponse pour traiter une requête pour chaque serveur utilisé séparément. La ligne « Avec l'IDS » du tableau donne la moyenne des temps de réponse de l'architecture pour traiter une requête. La ligne « Latence engendrée par l'IDS » donne le coût en terme de latence de l'introduction de l'IDS et de l'architecture en moyenne.

	IIS	thttpd	Apache
Sans l'IDS	0.0214s	0.0173s	0.0183s
Avec l'IDS	0.1256s		
Latence engendrée par l'IDS	0.1042s	0.1083s	0.1073s

TABLE 4.8 – Moyenne des latences en présence de l'IDS et en son absence.

En activant le proxy et l'IDS de type boîte noire, comme expliqué en 2.1.6, de nombreuses communications supplémentaires sont introduites dans le chemin de traitement de la requête : en moyenne, cela multiplie par 6 le temps de traitement d'une requête. Le temps de traitement reste acceptable malgré la latence introduite par l'architecture. Il faut rappeler que nous n'avons développé qu'un prototype qui est loin d'être optimisé. De plus, nous avons proposé des solutions en 2.1.6.2 pour diminuer le temps de latence au prix d'une diminution des capacités de tolérance aux intrusions.

4.6.2 Performance de l'IDS de type boîte grise

Comme nous l'avons expliqué en décrivant le modèle de détection et l'algorithme de calcul de similarité entre les graphes de flux d'information en 2.3.3.2, le temps de calcul est exponentiel en fonction du nombre de nœuds des deux graphes. Nous avons cherché à optimiser ce calcul de similarité qui nécessite de calculer tous les mappings possibles en limitant les mappings unitaires à ceux ayant un sens du point de vue des graphes considérés qui sont des graphes de flux d'informations. 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 5 minutes pour des graphes de 15 nœuds par exemple. Notre prototype d'IDS de type boîte grise ne peut donc être utilisé en temps réel actuellement. Nous l'avons donc

utilisé en système de détection d'intrusions a posteriori dans nos tests : il n'est pas sur le chemin de traitement de la requête.

Pour améliorer les performances, il nécessaire, d'une part, d'évaluer d'autres algorithmes tels que l'algorithme glouton proposé par Champin et Solnon. Il faudrait vérifier que les approximations faites par cet algorithme n'entraînent pas de faux positifs ou de faux négatifs. Il est d'autre part nécessaire d'optimiser le code de notre prototype, écrit également en *perl*.

4.7 Résumé

Dans ce chapitre, nous avons montré les résultats obtenus par nos deux prototypes d'IDS : l'un suivant une approche de type boîte noire et l'autre suivant une approche de type boîte grise. Nous avons étudié différents aspects de ces IDS : leurs performances en terme de détection d'intrusions par une évaluation de leur fiabilité (taux de faux négatifs) et leur pertinence (taux de faux positifs).

Nous avons d'abord décrit les deux environnements de tests : serveur web statique et serveur web dynamique, puis les différents jeux de données que nous avons utilisé pour ces deux environnements : leur provenance pour le trafic servant à évaluer la pertinence (logs de serveurs web, trafic « artificiel » sauvegardé) et la description des attaques que nous avons jouées contre notre architecture.

Nous avons ensuite présenté les résultats de détection obtenus en utilisant ces différents jeux de données pour étudier la pertinence et la fiabilité des deux IDS. Les résultats sont corrects bien que le taux de faux positifs soit légèrement élevé spécialement pour l'IDS de type boîte grise. La mise en place de solutions pour masquer les différences de spécification semble nécessaire à ce niveau, éventuellement grâce à un apprentissage des correspondances entre les graphes.

Par la suite, nous avons présenté les capacités d'aide au diagnostic que l'IDS de type boîte grise apporte à l'administrateur sur deux exemples d'intrusions : dans le premier exemple, l'aide au diagnostic est flagrante car les flux et objets non associés dans les meilleurs mappings correspondent exactement à l'objectif de l'intrusion. Dans le second exemple, les objets et flux non associés dans les meilleurs mappings montrent les effets de l'intrusion de manière indirecte et l'aide au diagnostic est moins efficace.

Finalement, nous avons présenté les performances en termes de latence et de calcul des similarités de nos prototypes. Elles sont moyennes mais il faut noter que nos prototypes sont loin d'être optimisés en terme d'algorithmique et de programmation.

Conclusion

« *Cattle die, Kinsmen die,*
Even you yourself leave this world,
But one thing I know that never dies,
Judgement on a dead man. »
Hávamál.

L'objectif de cette thèse est la détection d'intrusions affectant des logiciels de type serveur. Les approches de détection par signature ayant des défauts inhérents tels que l'impossibilité de détecter des intrusions inconnues, nous avons proposé une approche de détection comportementale. Contrairement aux méthodes de détection classiques en détection comportementale pour lesquelles il est nécessaire de définir et construire un modèle de référence du comportement de l'entité surveillée, nous avons suivi une méthode issue de la sûreté de fonctionnement fondée sur la programmation N-versions pour laquelle le modèle de référence est implicite et est constitué par les autres logiciels constituant l'architecture. Contrairement à la programmation N-versions classique où les versions sont développées spécifiquement, ce qui a un coût important, réservant ainsi cette technique à des projets critiques du point de vue de la sécurité-innocuité, nous proposons l'utilisation de COTS à la place des versions. C'est un choix fait dans de nombreux projets. Cependant, peu d'autres travaux de recherche ont mis en valeur les contraintes et problèmes qu'impose l'utilisation de COTS dans une telle architecture.

Nous proposons ici un résumé de l'ensemble de notre travail puis terminons par quelques perspectives.

Résumé

Le chapitre 2 présente le cœur de nos travaux. Nous avons utilisé une architecture de détection d'intrusions fondée sur la diversification de COTS, qui reste très similaire à celle d'autres projets. Nous en avons étudié différents aspects pour mettre en évidence les avantages et les limites d'une approche de détection fondée sur la diversification de COTS. Nous avons formalisé les propriétés de détection d'intrusions, de localisation des serveurs compromis et de tolérance aux intrusions dans le cadre de la diversification de COTS en distinguant deux cas : observations parfaites ou non. Nous avons ensuite présenté les deux ap-

proches de détection d'intrusions, que nous avons proposées, fondées sur cette architecture.

Notre approche de type boîte noire se distingue de l'existant par un algorithme de détection précis et une méthode de masquage des différences de conception ou de spécification différente, qui permet de limiter les faux positifs même dans des cas où le comportement des COTS est différent ou mal défini.

Notre approche de type boîte grise repose sur le calcul de similarité entre graphes de flux d'informations au niveau du système d'exploitation. Nous avons présenté le modèle et l'algorithme de calcul de la similarité. Nous avons discuté les choix faits lors de l'utilisation de ce modèle dans le cadre de la diversification de COTS pour la détection et la tolérance aux intrusions. En plus de ces capacités de détection, il est possible d'utiliser cette méthode pour aider l'administrateur à diagnostiquer les anomalies détectées.

Le chapitre 3 présente la manière dont nous avons appliqué le modèle général de détection d'intrusions au cadre des serveurs web. Nous avons décrit plus précisément le protocole HTTP qui est un protocole léger et universel. Ses caractéristiques permettent d'expliquer les quelques difficultés que doit résoudre notre algorithme de comparaison des réponses réseau. Nous avons décrit précisément cet algorithme de comparaison et détaillé les mécanismes de masquage des différences de spécification que nous avons mis en place : normalisation des requêtes et masquage des différences de sorties connues dans le cadre des serveurs web.

Nous avons présenté également notre IDS par calcul de la similarité entre graphes de flux d'informations. Les flux d'informations sont obtenus grâce à l'interception des appels système. Ces appels système ne sont cependant pas suffisants pour obtenir tous les flux d'informations. Les autres méthodes pour les obtenir semblent entraîner une baisse de performance trop importante cependant. Nous avons détaillé le processus de création des graphes de flux d'informations et exposé quelques choix concernant le calcul de la similarité entre les graphes. Nous avons présenté notre architecture combinant les deux IDS.

Finalement dans ce chapitre, nous avons exposé le problème posé par l'exécution d'applications web au-dessus des serveurs web et une solution pour résoudre partiellement ce problème, solution qui permet de détecter les injections de code de haut-niveau.

Le chapitre 4 présente les tests que nous avons menés et les résultats obtenus par nos deux prototypes d'IDS pour serveurs web. Nous avons évalué différents aspects de ces IDS : leurs performances en terme de détection d'intrusions par une évaluation de leur fiabilité (taux de faux négatifs) et leur pertinence (taux de faux positifs).

Nous avons d'abord décrit les deux environnements de tests : serveur web statique et serveur web dynamique, puis les différents jeux de données que nous avons utilisés pour ces deux environnements : leur provenance pour le trafic servant à évaluer la pertinence (logs de serveurs web, trafic « artificiel » sauvegardé) et la description des attaques que nous avons jouées contre notre architecture.

Nous avons ensuite présenté les résultats de détection obtenus en utilisant ces différents jeux de données pour étudier la pertinence et la fiabilité des deux IDS. Les résultats sont corrects : les intrusions jouées sur l'architecture sont toutes détectées à l'exception des intrusions par *cross-site scripting*. Le taux de faux positifs de notre IDS de type boîte noire est raisonnable : inférieur à 1%. Le taux de faux positifs de notre IDS de type boîte grise est plus élevé. La mise en place de solutions pour masquer les différences de spécification semble nécessaire à ce niveau, éventuellement grâce à un apprentissage des correspondances entre les graphes.

Par la suite, nous avons présenté les capacités d'aide au diagnostic que l'IDS de type boîte grise apporte à l'administrateur sur deux exemples d'intrusions : dans le premier exemple, l'aide au diagnostic est flagrante car les flux et objets non associés dans les meilleurs mappings correspondent exactement à l'objectif de l'intrusion. Dans le second exemple, les objets et flux non associés dans les meilleurs mappings montrent les effets de l'intrusion de manière indirecte et l'aide au diagnostic est moins directe.

Finalement, nous avons présenté les performances en termes de latence et de calcul des similarités de nos prototypes. Elles sont moyennes. Notre IDS de type boîte noire est utilisable en ligne, c'est-à-dire placé sur le chemin de la requête bien que le temps de traitement de la requête augmente significativement. Il n'est pas, par contre, possible d'utiliser notre IDS de type boîte grise en temps réel : le temps de calcul de la similarité entre les graphes peut être de l'ordre de la dizaine de minutes si les graphes ont de nombreux nœuds. Il faut, cependant, noter que nos prototypes sont loin d'être optimisés en terme d'algorithmique et de programmation.

Perspectives

Notre travail ouvre des perspectives donnant lieu à des études complémentaires dans les directions suivantes :

- Il serait intéressant d'évaluer cette méthode de détection dans un autre cadre que celui des serveurs web comme par exemple, celui des serveurs FTP, POP3 ou SIP. La plupart des travaux fondés sur l'approche par diversification de COTS, ainsi que les nôtres, ont choisi le cadre des serveurs web. Des études dans un autre cadre pourraient confirmer la généralité de l'approche ; il est notamment nécessaire de vérifier que les propriétés des COTS : indépendance des vulnérabilités et spécification suffisamment proche. La mise en œuvre de cette méthode du côté client, par exemple dans des navigateurs web, dans le cadre d'une approche boîte grise pourrait éventuellement être intéressante en termes de tolérance aux intrusions, étant donné que de nombreuses vulnérabilités ont été découvertes dans les applications clientes récemment.
- Dans le cadre de notre approche de type boîte noire, il peut être intéressant de modifier les choix de détection : nous avons en effet choisi de ne pas comparer certaines parties des réponses quand elles ne sont pas spécifiées dans la spécification standardisée. Cela laisse la possibilité à l'attaquant

de faire sortir des informations confidentielles. La modélisation statistique de ces parties des réponses est, par exemple, une solution envisageable pour assurer une couverture de détection plus importante.

– Notre approche de type boîte grise comporte de nombreux paramètres qu'il serait intéressant d'évaluer plus en avant : nous avons proposé plusieurs méthodes pour décider quand lever une alerte ou non, plusieurs règles de localisation, la prise en compte de différents seuils suivant le type de la requête. Il serait intéressant également d'évaluer l'influence d'un algorithme glouton sur les résultats de détection. L'avancée la plus importante serait de prendre en compte les données des flux d'informations. Nous avons limité cette idée aux flux sortant sur la socket vers le client en prenant en compte la réponse de l'IDS de type boîte noire. La prise en compte des données dans tous ou au moins une grande majorité des flux d'informations permettrait une plus grande précision dans l'algorithme de calcul de la similarité et ainsi une meilleure fiabilité de l'IDS. Éventuellement, les mêmes problèmes que ceux posés par les sorties réseau peuvent se poser au niveau de la comparaison de ces flux d'informations et il faudrait introduire des algorithmes de comparaison prenant en compte les différences de conception et de spécification.
– Dans le cadre de notre application pour les serveurs web, nous avons proposé une méthode pour assurer la diversification des langages de script dans lesquels sont écrits une grande partie des applications web. Cette méthode permet de détecter les tentatives d'injection de code. Une implémentation automatique de cette méthode est nécessaire pour évaluer la faisabilité de l'approche.
– Les tests que nous avons menés ne l'ont été que dans des environnements contrôlés. Il serait souhaitable de tester nos propositions dans un environnement réel. Cela n'est pas évident puisque nos IDS sont intimement liés au service rendu. Une utilisation de type « Honeypot » pourrait éventuellement servir à évaluer la fiabilité de l'approche dans un contexte réel. Si l'on souhaite conserver les propriétés de tolérance aux intrusions, l'évaluation de la pertinence dans un environnement réel semble bien plus problématique car les faux positifs entraînent des interruptions du service qui n'auraient pas lieu d'être.
– Les méthodes que nous avons proposées ne prennent pas en compte la politique de sécurité : ce sont des méthodes comportementales où le comportement de référence est le comportement des autres COTS. Elles se rapprochent dans ce sens des approches de détection d'intrusions fondées sur la spécification. La combinaison ou la corrélation des nos IDS avec d'autres IDS prenant en compte la politique de sécurité pourrait améliorer la fiabilité et la pertinence voire le diagnostic des alertes.

Bibliographie

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

[ABB+00] Jean Arlat, Jean-Paul Blanquart, Thierry Boyer, Yves Crouzet, Marie-Hélène Durand, Jean-Charles Fabre, Michel Founau, Mohamed Kaâniche, Karama Kanoun, Philippe Le Meur, Corinne Mazet, David Powell, François Scheerens, Pascale Thévenod-Fosse, and Hélène Waeselynck. *Composants logiciels et sûreté de fonctionnement : intégration de COTS*. Hermes Science Publications, juin 2000.

[ABC+95] Jean Arlat, Jean-Paul Blanquart, Alain Costes, Yves Crouzet, Yves Deswarte, Jean-Charles Fabre, Hubert Guillermain, Mohamed Kaâniche, Karama Kanoun, Jean-Claude Laprie, Corinne Mazet, David Powell, Christophe Rabéjac, and Pascale Thévenod. *Guide de la Sûreté de Fonctionnement*. Laboratoire d'Ingénierie de la Sûreté de fonctionnement (LIS), 1995.

[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.

[ADD00] Magnus Almgren, Hervé Debar, and Marc Dacier. A lightweight tool for detecting web server attacks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'2000)*, pages 157–170, San Diego, CA, February 2000.

[AK84] Algirdas Avizienis and John P. J. Kelly. Fault tolerance by design diversity : Concepts and experiments. *IEEE Computer*, 17 :67–80, August 1984.

[AL81] Tom Anderson and P.A. Lee. *Fault Tolerance, Principles and Practice*. Prentice Hall, 1981.

[AL01] Magnus Almgren and Ulf Lindqvist. Application-integrated data collection for security monitoring. In *Proceedings of the fourth International Symposium on Recent Advances in Intrusion Detection (RAID 2001)*, pages 22–36, Davis, CA, October 2001.

[ALJ+95] Debra Anderson, Teresa F. Lunt, Harold S. Javitz, Ann Tamaru, and Alfonso Valdes. Detecting unusual program behavior using the statistical components of nides. Technical Report SRI-CSL-95-06, SRI International, Computer Science Laboratory, Menlo Park, CA, May 1995.

[ALS88] Algirdas Avizienis, Michael R. Lyu, and Werner Schutz. In search of effective diversity : a six-language study of fault-tolerant flight control software. In *Digest of 18th FTCS*, pages 15–22, Tokyo, Japan, June 1988.

[And80] James P. Anderson. Computer security threat monitoring and surveillance. Technical report, James P. Anderson Company, Fort Washington, Pennsylvania, April 1980.

[Ara03] Arachnids : advanced reference archive of current heuristics for network intrusion detection systems, 2003. `http://www.whitehats.com/ids`.

[Avi67] Algirdas Avizienis. Design of fault-tolerant computers. In *Proceedings of the AFIPS Fall Joint Computer Conference*, volume 31, pages 733–743, Washington, D.C., November 1967.

[Avi95] Algirdas Avizienis. *The Methodology of N-Version Programming*, chapter 2, pages 23–46. Wiley, 1995.

[Axe00a] Stefan Axelsson. The base-rate fallacy and the difficulty of intrusion detection. *ACM Transactions on Information and System Security (TISSEC)*, 3(3) :186–205, 2000.

[Axe00b] Stefan Axelsson. Intrusion detection systems : A taxonomy and survey. Technical Report 99-15, Dept. of Computer Engineering, Chalmers University of Technology, March 2000.

[BAF+03] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM conference on Computer and communications security (CCS'03)*, pages 281–289, Washington, D.C., USA, October 2003.

[BDS03] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation : an efficient approach to combat a board range of memory error exploits. In *Proceedings of the 12th conference on USENIX Security Symposium*, pages 105–120, Washington, DC, USA, August 2003.

[BK04] Stephen W. Boyd and Angelos D. Keromytis. SQLrand : Preventing SQL injection attacks. In *Proceedings of the second International Conference on Applied Cryptography and Network Security (ACNS 2004)*, pages 292–302, Yellow Mountain, China, June 2004.

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

[BP76] David E. Bell and Leonard J. La Padula. Secure computer system : Unified exposition and multics interpretation. Technical report, MITRE Co., March 1976.

[BS08] Sandeep Bhatkar and R. Sekar. Data space randomization. In *Proceedings of the 5th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'08)*, pages 1–22, Paris, France, July 2008.

[BV05] Edward Balas and Camilo Viecco. Towards a third generation data capture architecture for honeynets. In *Proceedings from the Sixth Annual IEEE Systems, Man and Cybernetics (SMC) Information Assurance Workshop*, pages 21–28, West Point, NY, June 2005.

[BZ06] Emery D. Berger and Benjamin G. Zorn. Diehard : probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation (PLDI'06)*, pages 158–168, Ottawa, Ontario, Canada, June 2006.

[BZ07] Emery D. Berger and Benjamin G. Zorn. Efficient probabilistic memory safety. Technical Report Technical Report UMCS TR-2007-17, Department of Computer Science, University of Massachusetts Amherst, March 2007.

[CA78] L. Chen and Algirdas Avizienis. N-version programming : A fault tolerance approach to reliability of software operation. In *Proceedings of the 8th International Symposium on Fault-Tolerant Computing (FTCS-8)*, pages 3–9, Toulouse, France, June 1978.

[CEF+06] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant systems : A secretless framework for security through diversity. In *Proceedings of the 15th USENIX Security Symposium*, Vancouver, Canada, August 2006.

[CO00] Frédéric Cuppens and Rodolphe Ortalo. LAMBDA : A Language to Model a Database for Detection of Attacks. In H. Debar, L. Mé, and S. F. Wu, editors, *Proceedings of the Third International Workshop on the Recent Advances in Intrusion Detection*

(RAID'2000), number 1907 in LNCS, pages 197–216, October 2000.

[CRL03] Miguel Castro, Rodrigo Rodrigues, and Barbara Liskov. Base : Using abstraction to improve fault tolerance. *ACM Trans. Comput. Syst.*, 21(3) :236–269, August 2003.

[CS02] Monica Chew and Dawn Song. Mitigating buffer overflows by operating system randomization. Technical Report CMU-CS-02-197, Carnegie Mellon University, December 2002.

[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.

[d'A94] Bruno d'Ausbourg. Implementing secure dependencies over a network by designing a distributed security subsystem. In *Proceedings of the European Sysmposium on Research in Computer Security (ESORICS'94)*, pages 249–266, Brighton, UK, November 1994.

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

[DFH96] Patrik D'haeseleer, Stephanie Forrest, and Paul Helman. An immunological approach to change detection : Algorithms, analysis and implications. In *Proceedings of the 1996 IEEE Symposium on Research in Security and Privacy*, pages 110–119, Oakland, CA, may 1996. IEEE Computer Society, IEEE Computer Society Press.

[DKL99] Yves Deswarte, Karama Kanoun, and Jean-Claude Laprie. Diversity against accidental and deliberate faults. In P.Ammann, B.H.Barnes, S.Jajodia, and E.H.Sibley, editors, *Computer Security, Dependability, & Assurance : from needs to solutions*, pages 171–181, Los Alamitos, CA, 1999. IEEE Computer Society.

[DT05] Hervé Debar and Elvis Tombini. Webanalyzer : Accurate and fast detection of http attack traces in web server logs. In *Proceedings of EICAR 2005*, Malta, 2005.

[Edd07] Wesley M. Eddy. Tcp syn flooding attacks and common mitigations, August 2007. RFC 4987.

[EL85] Dave E. Eckhardt and Larry D. Lee. A theoretical basis for the analysis of multiversion software subject to coincident errors. *IEEE Transactions on Software Engineering (TSE)*, 11(12) :1511–1517, December 1985.

[Elm72] William R. Elmendorf. Fault-tolerant programming. In *Proceedings of the 2nd International Symposium on Fault-Tolerant*

Computing (FTCS-2), volume 31, pages 79–83, Newton, MA, June 1972.

[ETGTDV04] Juan M. Estévez-Tapiador, Pedro García-Teodoro, and Jesús E. Díaz-Verdejo. Measuring normality in HTTP traffic for anomaly-based intrusion detection. *Computer Networks*, 45(2) :175–193, June 2004.

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

[FGM+99] Roy Fielding, Jim Gettys, Jeff Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol - HTTP/1.1, June 1999. RFC 2616.

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

[Fie96] Roy Fielding. WWWstat, 1996. `http://ftp.ics.uci.edu/pub/websoft/wwwstat/`.

[FSA97] Stephanie Forrest, Anil Somayaji, and David H. Ackley. Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, pages 67–72, Cape Cod, MA, USA, May 1997.

[Gao07] Debin Gao. *Gray-Box Anomaly Detection using System Call Monitoring*. PhD thesis, Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, January 2007.

[GMS00] Anup K. Ghosh, Christoph Michael, and Michael Schatz. A real-time intrusion detection system based on learning program behavior. In *Proceedings of the Third International Workshop on the Recent Advances in Intrusion Detection (RAID'2000)*, pages 93–109, October 2000.

[GPS04] Ilir Gashi, Peter Popov, and Lorenzo Strigini. Fault diversity among off-the-shelf SQL database servers. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '04)*, pages 389–398, Florence, Italy, 2004.

[GPSS04] Ilir Gashi, Peter Popov, Vladimir Stankovic, and Lorenzo Strigini. *On Designing Dependable Services with Diverse Off-The-Shelf SQL Servers*, volume 3069 of *Lecture Notes in Computer Science*, pages 196–220. Springer-Verlag, 2004.

[Gra85] Jim Gray. Why do computers stop and what can be done about it ? Technical Report Technical Report 85.7, TandemComputers, Cupertino, California, USA, June 1985.

[Gra86] Jim Gray. Why do computers stop and what can be done about it ? In *Proceedings of the 5th Symposium on Reliability in Distributed Systems and Database Systems (SRDSDS-5)*, pages 3–12, Los Angeles, California, USA, January 1986.

[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.

[GRS06a] Debin Gao, Michael K. Reiter, and Dawn Song. Behavioral distance measurement using hidden markov models. In *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID 2006)*, pages 19–40, Hamburg, Germany, September 2006.

[GRS06b] Debin Gao, Michael K. Reiter, and Dawn Song. Beyond output voting : Detecting compromised replicas using behavorial distance. Technical Report CMU-CyLab-06-019, CyLab - Carnegie Mellon University, December 2006.

[HDL+90] L. Todd Heberlein, Gihan Dias, Karl N. Levitt, Biswanath Mukherjee, Jeff Wood, and David Wolber. A network security monitor. In *Proceedings ot the 1990 IEEE Computer Society Symposiumon Research in Security and Privacy*, pages 296–304, Oakland, CA, May 1990.

[HK88] Lawrence R. Halme and Brian L. Kahn. Building a security monitor with adaptive user work profiles. In *Proceedings of the 11th National Computer Security Conference*, pages 274–283, Washington, D.C., October 1988.

[HL93] Paul Helman and Gunar E. Liepins. Statistical foundations of audit trail analysis for the detection of computer misuse. *IEEE Transactions on Software Engineering*, 19(9) :886–901, September 1993.

[HNM+06] Michel Hurfin, Jean-Pierre Le Narzul, Frédéric Majorczyk, Ludovic Mé, Ayda Saidane, Eric Totel, and Frédéric Tronel. A dependable intrusion detection architecture based on agreement services. In *Proceedings of the Eighth International Symposium on Stabilization, Safety, and Security of Distributed Systems*, Dallas, TX, November 2006.

[II07] Kenneth L. Ingham and Hajime Inoue. Comparing anomaly detection techniques for http. In *Proceeding of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID'2007)*, pages 42–62, Queensland, Australia, September 2007. Springer.

[Ing07] Kenneth L. Ingham. *Anomaly Detection for HTTP Intrusion Detection : Algorithm Comparisons and the Effect of Genera-*

lization on Accuracy. PhD thesis, University of New Mexico, Albuquerque, New Mexico, May 2007.

[ISBF07] Kenneth L. Ingham, Anil Somayaji, John Burge, and Stephanie Forrest. Learning DFA representations of HTTP for protecting web applications. *Computer Networks*, 51(5) :1239–1255, 2007.

[JA88] Mark K. Joseph and Algirdas Avizienis. A fault tolerant approach to computer viruses. In *Proceedings of the IEEE Symposium on Security and Privacy (SSP'88)*, pages 52–58, 1988.

[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.

[KAU+86] John P. J. Kelly, Algirdas Avizienis, Bradford T. Ulery, B. J. Swain, Michael R. Lyu, Ann T. Tai, and Kam S. Tso. Multi-version software development. In *Proceedings of IAFC Qorkshop SAFECOMP'86*, pages 43–49, Sarlat, France, October 1986.

[KC03] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 223–236, Bolton Landing, NY, October 2003. ACM Press.

[KC05] Samuel T. King and Peter M. Chen. Backtracking intrusions. *ACM Transactions on Computer Systems*, 23(1) :51–76, February 2005.

[KKP03] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security (CCS'03)*, pages 272–280, Washington, DC, October 2003.

[KRV04] Christopher Kruegel, William K. Robertson, and Giovanni Vigna. Using alert verification to identify successful intrusion attempts. *Practice in Information Processing and Communication (PIK)*, 27(4), October 2004.

[KTK02] Christopher Kruegel, Thomas Toth, and Engin Kirda. Service specific anomaly detection for network intrusion detection. In *Proceedings of the Symposium on Applied Computing (SAC 2002)*, pages 201–208, Madrid, Spain, March 2002.

[KV03] Christopher Kruegel and Giovanni Vigna. Anomaly detection of web-based attacks. In *Proceedings of the 10th ACM Conference on Computer and Communication Security (CCS'03)*, pages 251–261, Washington, D. C., October 2003. ACM Press.

[KVR05] Christopher Kruegel, Giovanni Vigna, and William K. Robertson. A multi-model approach to the detection of web-based attacks. *Computer Networks*, 48(5) :717–738, August 2005.

[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.

[LHF+00] Richard Lippmann, Joshua W. Haines, David J. Fried, Jonathan Korba, and Kumar Das. The 1999 darpa off-line intrusion detection evaluation. *Computer Networks*, 34(4) :579–595, October 2000.

[LM89] Bev Littlewood and Douglas R. Miller. Conceptual modeling of coincident failures in multiversion software. *IEEE Transactions on Software Engineering (TSE)*, 15(12) :1596–1614, December 1989.

[LS04] Bev Littlewood and Lorenzo Strigini. Redundancy and diversity in security. In *Proceedings of the 9th European symposium on research in computer security (ESORICS 04)*, pages 423–438, Sophia Antipolis, September 2004.

[LTG+90] Teresa F. Lunt, Ann Tamaru, Fred Gilham, R. Jagannathan, Caveh Jalali, Harold S. Javitz, Alfonso Valdes, and Peter G. Neumann. A real-time intrusion-detection expert system. Technical report, SRI International, June 1990.

[Lun88] Teresa F. Lunt. Automated audit trail analysis and intrusion detection : a survey. In *Proceedings of the 11th National Computer Security Conference*, pages 65–73, Washington, D.C., October 1988.

[McH01] John McHugh. Intrusion and intrusion detection. *International Journal of Information Security*, July 2001.

[Mck84] A. Mckenzie. ISO transport protocol specification. ISO DP 8073, April 1984. RFC 905.

[MHL94] Biswanath Mukherjee, L. Todd Heberlein, and Karl N. Levitt. Network intrusion detection. *IEEE Network*, 8(3) :26–41, May-June 1994.

[MHL+03] Peter Mell, Vincent Hu, Richard Lippmann, Josh Haines, and Marc Zissman. An overview of issues in testing intrusion detection. http ://csrc.nist.gov/publications/nistir/nistir-7007.pdf, 2003.

[MM01] Cédric Michel and Ludovic Mé. Adele : an attack description language for knowledge-based intrusion detection. In *Proceedings of the 16th International Conference on Information Security (IFIP/SEC 2001)*, pages 353–365, June 2001.

[MPS+03] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the slammer worm. *Security and Privacy*, 1(5) :33–39, September-October 2003.

[MTM05] Frédéric Majorczyk, Eric Totel, and Ludovic Mé. Détection d'intrusions par diversification de cots. In *Proceedings the 4th Conference on Security and Network Architectures (SAR'2005)*, Batz-sur-Mer, France, June 2005.

[MTM07] Frédéric Majorczyk, Eric Totel, and Ludovic Mé. Experiments on cots diversity as an intrusion detection and tolerance mechanism. In *Proceedings of the First Workshop on Recent Advances on Intrusion-Tolerant Systems (WRAITS 2007)*, Lisbon, Portugal, March 2007.

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

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

[Mye80] Philip Myers. Subversion : The neglected aspect of computer security. Master's thesis, Naval Postgraduate School, June 1980.

[Nav01] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Transactions on Computer Systems*, 33(1) :31–88, 2001.

[NBZ07] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Exterminator : Automatically correcting memory errors with high probability. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI'07)*, pages 1–11, San Diego, California, USA, June 2007.

[NS05] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS 2005)*, San Diego, CA, February 2005.

[NTEK+08] Anh Nguyen-Tuong, David Evans, John C. Knight, Benjamin Cox, and Jack W. Davidson. Security through redundant data diversity. In *Proceedings of the 38th IEEE/IFPF International Conference on Dependable Systems and Networks (DSN 2008)*, Anchorage, Alaska, USA, June 2008.

[PABD+99] David Powell, Jean Arlat, Ljerka Beus-Dukic, Andrea Bondavalli, Paolo Coppola, Alessandro Fantechi, Eric Jenn, Christophe Rabéjac, and Andy J. Wellings. GUARDS : A generic upgradable architecture for real-time dependable systems. *IEEE Transactions On Parallel and Distributed Systems*, 10(6) :580–599, 1999.

[Pax98] Vern Paxson. Bro : A system for detecting network intruders in real-time. In *Proceedings of the 7th Usenix Security Symposium*, pages 31–51, San Antonio, TX, January 1998.

[PMRM08] Niels Provos, Panayiotis Mavrommatis, Moheeb A. Rajab, and Fabian Monrose. All your iFRAMEs point to us. Technical report, Google Inc, 1600 Amphitheatre Parkway, Mountain View, CA, February 2008.

[PN97] Phillip A. Porras and Peter G. Neumann. EMERALD : Event monitoring enabling responses to anomalous live disturbances. In *Proc. of the 20th National Information Systems Security Conference*, pages 353–365, Baltimore, MD, October 1997.

[Pro03] The Honeynet Project. Know your enemy : Sebek, November 2003.

[PS03] Peter Popov and Lorenzo Strigini. Diversity with off-the-shelf components a study with sql database servers. In *DSN 2003, International Conference on Dependable Systems and Networks*, pages B84–B85, San Francisco, CA, June 2003.

[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.

[RCL01] Rodrigo Rodrigues, Miguel Castro, and Barbara Liskov. Base : Using abstraction to improve fault tolerance. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, pages 15–28, Chateau Lake Louise, Canada, October 2001.

[RD86] Brian Randell and John E. Dobson. Reliability and security issues in distributed computing systems. In *Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems*, pages 113–118, Los Angeles, CA, USA, January 1986.

[Ris08] Ivan Ristic. ModSecurity 2.5, 2008. `http://www.modsecurity.org/`.

[Riv92] Ronald L. Rivest. The md5 message-digest algorithm, April 1992. RFC 1321.

[RJCM03] James C. Reynolds, James Just, Larry Clough, and Ryan Maglich. On-line intrusion detection and attack prevention using diversity, generate-and-test, and generalization. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03)*, page 335.2, Big Island, HI, January 2003. IEEE Computer Society.

[RJL+02] James C. Reynolds, James E. Just, Ed Lawson, Larry A. Clough, Ryan Maglich, and Karl N. Levitt. The design and implementation of an intrusion tolerant system. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 285–292, Bethesda, MD, USA, June 2002.

[RM08] Inez Raquenet and Carlos Maziero. A fuzzy model for the composition of intrusion detectors. In *Proceedings of the 23rd IFIP International Information Security Conference (IFIP SEC 2008)*, pages 237–251, Milano, Italy, September 2008.

[Roe99] Martin Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the USENIX LISA'99 conference*, pages 229–238, Seattle, WA, November 1999.

[RVKK06] William K. Robertson, Giovanni Vigna, Christopher Kruegel, and Richard A. Kemmerer. Using generalization and characterization techniques in the anomaly-based detection of web attacks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2006)*, San Diego, CA, February 2006.

[SDN03] Ayda Saidane, Yves Deswarte, and Vincent Nicomette. An intrusion tolerant architecture for dynamic content internet servers. In Peng Liu and Partha Pal, editors, *Proceedings of the 2003 ACM Workshop on Survivable and Self-Regenerative Systems (SSRS-03)*, pages 110–114, Fairfax, VA, October 2003. ACM Press.

[Sec08] Breach Security. WebDefend, 2008. `http://www.breach.com/products/webdefend.html`.

[SEP05] Ana Nora Sovarel, David Evans, and Nathanael Paul. Where's the FEEB ? the effectiveness of instruction set randomization. In *Proceedings of the 14th conference on USENIX Security Symposium*, pages 145–160, Baltimore, MD, USA, August 2005.

[SGF08] Babak Salamat, Andreas Gal, and Michael Franz. Reverse stack execution in a multi-variant execution environment. In *Proceedings of the First Workshop on Compiler and Architectural Techniques for Application Reliability and Security (CATARS'08)*, Anchorage, Alaska, USA, June 2008.

[SGJ+08] Babak Salamat, Andreas Gal, Todd Jackson, Karthikeyan Manivannan, Gregor Wagner, and Michael Franz. Multi-variant program execution : Using multi-core systems to defuse buffer-overflow vulnerabilities. In *Proceedings of the Internationl Workshop on Multi-Core Computing Systems (MuCoCoS'08)*, Barcelona, Spain, March 2008.

[Tay99] Grant Taylor. Autobuse - log monitoring daemon, 1999.

[TDMD04] Elvis Tombini, Hervé Debar, Ludovic Mé, and Mireille Ducassé. A serial combination of anomaly and misuse IDSes applied to HTTP traffic. In *Proceedings of ACSAC'2004*, pages 428–437, Tucson, AZ, December 2004.

[Tho07] Yohann Thomas. *Policy-based response to intrusions through context activation.* PhD thesis, Ecole nationale supérieure des télécommunications de Bretagne, 2007.

[TKM02] Kymie M. C. Tan, Kevin S. Killourhy, and Roy A. Maxion. Undermining an anomaly-based intrusion detection system using common exploits. In *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID'2002)*, pages 54–73, 2002.

[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.

[Tra88] P. Traverse. Airbus and ATR system architecture and specification, 1988.

[Tri] Tripwire. http ://www.tripwire.org.

[Tro03] Frédéric Tronel. *Applications des problèmes d'accord à la tolérance aux défaillances dans les systèmes distribués asynchrones.* PhD thesis, Université de Rennes 1, 2003.

[VAC+02] Alfonso Valdes, Magnus Almgren, Steven Cheung, Yves Deswarte, Bruno Dutertre, Joshua Levy, Hassen Saïdi, Victoria Stravidou, and Tomas E. Uribe. An adaptive intrusion-tolerant server architecture. In *Proceedings of the 10th International Workshop on Security Protocols*, pages 158–178, Cambridge, United Kingdom, April 2002.

[vdMRSJ05] Meine J.P. van der Meulen, Steve Riddle, Lorenzo Strigini, and Nigel Jefferson. Protective wrapping of off-the-shelf components. In *Proceedings of the 4th International Conference on COTS-Based Software Systems (ICCBSS '05)*, pages 168–177, Bilbao, Spain, 2005.

[VL89] Hank S. Vaccaro and Gunar E. Liepins. Detection of anomalous computer session activity. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 280–289, Oakland, CA, May 1989.

[VNC03] Paulo E. Veríssimo, Nuno F. Neves, and Miguel P. Correia. Intrusion-tolerant architectures : Concepts and design. In *Architecting Dependable Systems*, volume 2677 of *Lecture Notes in Computer Science*. Sptringer-Verlag, April 2003.

[VRKK03] Giovanni Vigna, William K. 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.

[VVK03] Giovanni Vigna, Fredrik Valeur, and Richard A. Kemmerer. Designing and implementing a family of intrusion detection systems. In *Proceedings of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software*

Engineering (ESEC/FSE 2003), pages 88–97, Helsinki, Finland, September 2003.

[VVKK06] Fredrik Valeur, Giovanni Vigna, Christopher Kruegel, and Engin Kirda. An anomaly-driven reverse proxy for web applications. In *Proceedings of the 2006 ACM symposium on Applied computing (SAC '06)*, pages 361–368, Dijon, France, April 2006.

[WFMB03] Yu-Sung Wu, Bingrui Foo, Yongguo Mei, and Saurabh Bagchi. Collaborative intrusion detection system (cids) : A framework for accurate and efficient ids. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC'03)*, pages 234–244, Las Vegas, NV, USA, December 2003.

[WFP99] Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. Detecting intrusions using system calls : Alternative data models. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 133–145, Oakland, CA, May 1999.

[WGS+01] Feiyi Wang, Fengmin Gong, Chandramouli Sargor, Katerina Goseva-Popstojanova, Kishor Trivedi, and Frank Jou. Sitar : a scalable intrusion-tolerant architecture for distributed services. In *Proceedings of the 2001 IEEE Workshop on Information Assurance and Security*, pages 38–45, United States Military Academy, West Point, NY, June 2001.

[WMT03] Dazhi Wang, Bharat B. Madan, and Kishor S. Trivedi. Security analysis of sitar intrusion tolerance system. In *Proceedings of the 2003 ACM workshop on Survivable and self-regenerative systems : in association with 10th ACM Conference on Computer and Communications Security*, pages 23–32, Fairfax, VA, October 2003.

[WS02] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM conference on Computer and Communications Security (CCS'02)*, pages 255–264, Washington, DC, USA, November 2002.

[WS04] Ke Wang and Salvatore J. Stolfo. Anomalous payload-based network intrusion detection. In Erland Jonsson, Alfonso Valdes, and Magnus Almgren, editors, *Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection (RAID'2004)*, volume 3224 of *Lecture Notes in Computer Science*, pages 203–222. Springer, September 15-17 2004.

[WWB01] Rong Wang, Feiyi Wang, and Gregory T. Byrd. Design and implementation of acceptance monitor for building scalable intrusion tolerant system. In *Proceedings of the 10th International Conference on Computer Communications and Networks*, pages 200–5, Phoenix, AZ, October 2001.

[WWRS03] Ian Welch, John Wame, Peter Ryan, and Robert Stroud. Architectural analysis of MAFTIA's intrusion tolerance capabilities.

Technical Report Deliverable D99, MAFTIA Project, January 2003.

[WYF83] J. F. Williams, L. J. Yount, and J. B. Flannigan. Advanced autopilot flight director system computer architecture for boeing 737-300 aircraft. In *AIAA/IEEE 5th Digital Avionics Systems Conference*, Seattle, Washington, November 1983.

[XKI03] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent runtime randomization for security. In *Proceedings of the 22nd International Symposium Reliable Distributed Systems (SRDS'03)*, pages 260–269, Florence, Italy, October 2003.

Résumé

L'informatique et en particulier l'Internet jouent un rôle grandissant dans notre société. Un grand nombre d'applications critiques d'un point de vue de leur sécurité sont déployées dans divers domaines comme le domaine militaire, la santé, le commerce électronique, etc. La sécurité des systèmes informatiques devient alors une problématique essentielle tant pour les individus que pour les entreprises ou les états. Il est donc important de définir une politique de sécurité pour ces systèmes et de veiller à son respect. Néanmoins les mécanismes de sécurité préventifs mis en place ne sont pas incontournables. Il est nécessaire de mettre en œuvre des outils permettant de détecter toute violation de la politique de sécurité, c'est-à-dire toute intrusion. Ces outils sont appelés des systèmes de détection d'intrusions ou IDS. Souvent en collaboration avec les IDS, il est possible de mettre en place des outils et techniques de tolérance aux intrusions. Ces outils et techniques ont pour objectif que les intrusions affectant un composant du système n'entraînent pas de violations de la politique de sécurité du système global.

Notre travail s'inscrit dans le domaine de la détection d'intrusions, de manière essentielle, et permet une certaine tolérance aux intrusions. Contrairement aux méthodes de détection classiques en détection comportementale pour lesquelles il est nécessaire de définir et construire un modèle de référence du comportement de l'entité surveillée, nous avons suivi une méthode issue de la sûreté de fonctionnement fondée sur la programmation N-versions pour laquelle le modèle de référence est implicite et est constitué par les autres logiciels constituant l'architecture. Nous proposons l'utilisation de COTS en remplacement des versions spécifiquement développées car développer N versions est coûteux et est réservé à des systèmes critiques du point de vue de la sécurité-innocuité. D'autres travaux et projets ont proposé des architectures fondées sur ces idées.

Nos contributions se situent à différents niveaux. Nous avons pris en compte dans notre modèle général de détection d'intrusions les spécificités liées à l'utilisation de COTS en lieu et place de versions spécifiquement développées et proposé deux solutions pour parer aux problèmes induits par ces spécificités. Nous avons proposé deux approches de détection d'intrusions fondées sur cette architecture : l'une suivant une approche de type boîte noire et l'autre suivant une approche de type boîte grise. Notre méthode de type boîte grise peut, en outre, aider l'administrateur de sécurité à effectuer un premier diagnostic des alertes. Nous avons réalisé une implémentation de ces deux approches dans le cadre des serveurs web et avons évalué pratiquement la pertinence et de la fiabilité de ces deux IDS.

Mots Clés : sécurité des systèmes d'information, détection d'intrusions comportementale, diversification de COTS.

VU :
Le Directeur de Thèse

VU :
Le Responsable de l'École Doctorale

VU pour autorisation de soutenance
Rennes, le

Le Président de l'Université de Rennes 1

Monsieur Guy Cathelineau

VU après soutenance pour autorisation de publication :
Le Président du Jury,