Centre de recherches sur les communications Canada
Un organisme d'Industrie Canada

Communications Research Centre Canada
An Agency of Industry Canada

Systèmes et langages de détection d'intrusions

Mathieu Couture
Frédéric Massicotte

Rapport du CRC No : CRC-RP-2005-001
Ottawa, 27 juillet 2005

Centre de recherches sur les communications Canada
Un organisme d'Industrie Canada

Communications Research Centre Canada
An Agency of Industry Canada

Systèmes et langages de détection d'intrusions

Mathieu Couture
Frédéric Massicotte

Rapport du CRC No : CRC-RP-2005-001
Ottawa, 27 juillet 2005

Canada

CRC

Systèmes et langages de détection d'intrusions

Mathieu Couture
Frédéric Massicotte

Rapport du CRC No : CRC-RP-2005-001
Ottawa, 27 Juillet 2005

Résumé

Ce rapport consitue une revue de littérature sur les systèmes de détections d'intrusions à base de scénarios. Le focus est mis sur le langage qu'ils utilisent pour exprimer leurs signatures. Cinq catégories de langages ont été identifiées, parmi lesquelles nous avons classé chacun des treize systèmes étudiés. Une fois cette étude faite, nous identifions dix propriétés jugées comme souhaitables pour un langage de signatures utilisé par un système de détection d'intrusions.

Abstract

This report is a state of the art about scenario based intrusion detection systems. The focus is on the language used to express their signatures. Five categories have been identified, among which we have classified each of the thirteen studied systems. Once this done, we identify ten properties considered desirable for an intrusion detection system signature language.

Sommaire

Les systèmes de détection d'intrusions se divisent en deux catégories : détecteurs d'anomalies et systèmes à base de signatures, aussi appelés systèmes à base de scénarios. Les détecteurs d'anomalies fonctionnent en deux phases : une phase d'apprentissage et une phase de détection. Durant la phase d'apprentissage, le système se crée un modèle de comportement considéré comme normal, et la phase de détection consiste à identifier des déviations de ce modèle. Les systèmes à base de signatures, quant à eux, fonctionnent à partir d'une base de descriptions d'attaques connues. L'avantage des détecteurs d'anomalies est leur capacité à détecter des attaques inconnues, alors que celui des systèmes à base de signatures est leur faible taux de faux-positifs.

Dans [37], Cédric Michel et Ludovic Mé proposent une taxonomie des langages utilisés en détection d'intrusions divisant ceux-ci en six catégories : langages d'événements, d'exploits, de rapports, de détection, de corrélation et de réponse. Dans ce chapitre, nous nous attaquons au cas particulier des langages de détection, que nous subdivisons en cinq catégories. La première catégorie regroupe les langages de signatures spécifique à un domaine particulier : détection de paquets, détection passive de failles de sécurité, etc. La seconde catégorie est celle des langages impératifs, qui sont ceux qui se rapprochent le plus des langages de programmation réguliers. La troisième catégorie est celle des langages à base de systèmes de transition. Les langages dans cette catégorie sont construits à partir de paradigmes tels que les automates et les réseaux de Petri. Les langages de la quatrième catégorie, les systèmes experts, sont ceux où la notion d'inférence permet d'abstraire les différents événements afin de déduire des nouvelles informations. Finalement, les langages de la cinquième catégorie sont ceux basés sur des logiques temporelles.

À la fin du rapport, nous dressons une liste de dix propriétés identifiées comme souhaitables pour un langage de signatures utilisé par un système de détection d'intrusions. Ces dix propriétés sont : 1) capacité d'exprimer des scénarios à plusieurs événements, 2) capacité d'exprimer la non-occurence d'événements, 3) capacité d'exprimer des contraintes temps-réel, 4) supporter le comptage d'événements semblables, 5) permettre l'acquisition passive de connaissance, 6) permettre d'exprimer des propriétés n'étant vraies qu'entre deux événements donnés (ayant un début et une fin), 7) supporter la diversité des types d'événements pouvant survenir, 8) présenter une approche déclarative, 9) disposer d'un algorithme de vérification qui soit en-ligne et 10) n'utiliser qu'une quantité bornée d'espace mémoire pour fonctionner.

Table des matières

Table des figures

Liste des tableaux

Introduction

Un système de détection d'intrusions (ou IDS pour *Intrusion Detection System*) est une composante logicielle responsable d'identifier une utilisation non-désirable d'une ressource informatique. On peut classer les IDS selon plusieurs critères. Les critères les plus généralement rencontrés sont le type de sonde (réseau ou hôte) et le type d'algorithme (statistique ou à base de signatures).

Les systèmes basés sur des méthodes statistiques, aussi appelés détecteurs d'anomalies, présentent l'avantage de pouvoir découvrir des attaques encore non-répertoriées. Ils fonctionnent généralement en deux phases : une phase d'apprentissage et une phase de détection proprement dite. Pendant la phase d'apprentissage, le système dresse un profil de ce qui sera par la suite considéré comme un comportement normal. La phase de reconnaissance, quant à elle, consistera à détecter des déviations du comportement normal. Normalement, la phase d'apprentissage continue pendant la phase de reconnaissance. Cette stratégie permet de faire face à la nature changeante du comportement des utilisateurs et ainsi de s'adapter au changement. L'hypothèse derrière cette façon de faire est que le comportement normal d'un utilisateur change de façon graduelle et qu'une utilisation non-désirée du réseau entraînera un changement brusque dans le comportement. Cette hypothèse entraîne à la fois des faux-positifs (fausse alarme) et des faux-négatifs (attaque non-détectée). Les faux positifs surviennent lorsque les utilisateurs changent leur comportement de façon brusque, par exemple en installant un nouveau logiciel. Les faux négatifs, quant à eux, surviennent lorsqu'un utilisateur malveillant au courant de la stratégie de détection change son comportement petit à petit de façon à exploiter la capacité d'adaptation du système.

Les IDS fonctionnant à base de signatures, quant à eux, engendrent moins de faux-positifs car on leur spécifie exactement ce qu'ils doivent détecter. Cependant, comme on doit constamment tenir à jour leur base de signatures, le risque de faux-négatifs est plus élevé. Le modèle de système de détection d'intrusion à base de signatures le plus simple que nous puissions imaginer est celui généralement utilisé dans le domaine des anti-virus : celui du patron de bits. L'hypothèse menant à cette méthode de travail

est qu'un comportement non-désirable peut être détecté à partir d'un petit nombre de bits physiquement situés à proximité les uns des autres. Dans le cas des anti-virus, il s'agit généralement des premiers bits du fichier, alors que dans les IDS réseaux tels que Snort [19], il peut s'agir de quelques bits bien ciblés d'un seul paquet. Bien que naïve en apparence, la stratégie du patron de bits a mené jusqu'à date à de très bons résultats, et ce pour deux raisons : premièrement, la complexité algorithmique liée à la vérification des signatures est des plus avantageuses (constante en mémoire et linéaire en temps), deuxièmement, la simplicité du langage de signature encourage la communauté d'utilisateurs à participer à l'enrichissement de la base de données de signatures.

Les IDS fonctionnant à base de patrons de bits comportent cependant des faiblesses au niveau de l'expressivité des signatures qui peuvent mener autant à des faux-positifs qu'à des faux-négatifs. C'est pourquoi la communauté scientifique persiste à mettre tant d'efforts dans le développement d'IDS pouvant détecter des patrons d'événements séparés dans le temps. Les signatures de ces systèmes permettent de prendre en compte le contexte avant de signaler une alarme. Par exemple, dans le cas d'un IDS réseau, ils permettent de spécifier que pour être valide, une attaque doit survenir dans le contexte d'une session active. Différentes approches ont été utilisées dans le cadre du développement de ces systèmes, et ce rapport est consacré à leur étude.

Nous commencerons notre étude au chapitre 1 par les langages les plus simples : les langages spécifiques au domaine. Ces langages ont l'avantage de présenter une grande simplicité, mais offrent souvent une faible expressivité et peu de souplesse. Par la suite, au chapitre 2, nous jetterons un oeil aux langages impératifs ayant été développés spécialement pour la détection d'intrusion. Ce sont ceux qui se rapprochent le plus des langages de programmation habituels. Ils présentent l'avantage d'offrir des primitives associées au traitement d'événements et des systèmes de type appropriés à la détection d'intrusion. Les signatures développées avec ces langages sont cependant souvent difficiles à maintenir. Les langages basés sur les systèmes de transition, que nous traiterons au chapitre 3, ont été mis au point afin de pouvoir spécifier d'une façon déclarative les scénarios d'attaques. Ces langages comportent cependant une partie impérative dont l'objectif est souvent d'emmagasiner une partie du contexte. Les systèmes experts, dont nous parlerons dans le chapitre 4, sont spécialisés dans l'accumulation et, surtout, l'inférence d'information concernant le contexte. Afin d'éviter les problèmes de mémoire, cette information doit cependant être gérée explicitement, et le niveau d'abstraction désiré n'est pas toujours atteint. Les systèmes basés sur des logiques temporelles, dont nous parlerons finalement dans le chapitre 5, sont ceux qui se rapprochent le plus de systèmes à base de signatures purement déclaratives.

Chapitre 1

Langages spécifiques au domaine

Un langage spécifique au domaine, tel que défini dans [52], est un langage de programmation spécialement conçu pour capturer la sémantique propre à un domaine d'application particulier. Des exemples de langages spécifiques au domaine sont HTML et SQL, respectivement conçus pour l'édition d'hypertextes et la consultation de bases de données. Dans le présent chapitre, chacun des systèmes de détection d'intrusion que nous présentons vient avec son propre langage permettant de le configurer et/ou de spécifier des signatures particulières d'attaque. Ces langages, spécialement conçus pour la détection d'intrusion ou pour la surveillance de systèmes en général, tombent donc tous dans la catégorie des langages spécifiques au domaine. Cependant, dans la plupart des cas, ces langages sont conçus avec un certain souci d'abstraction permettant de les utiliser pour différents types de fichiers d'audit. Par exemple, le langage STATL peut être utilisé autant pour une détection d'intrusion au niveau hôte (dans le cas de WinSTAT et USTAT), qu'au niveau réseau (dans le cas de NetSTAT).

Les trois systèmes que nous présentons dans ce chapitre se distinguent de par le fait que les langages qu'ils utilisent sont liés d'encore plus près au type de détection particulier qu'ils permettent d'effectuer. Ils ne sont pas définis sur une sémantique abstraite de fichiers d'audit, mais sur des sémantiques très concrètes reliées aux fichiers d'audits de processus (dans le cas de Panoptis), au filtrage de paquets (dans le cas de Snort), et à l'identification passive de failles de sécurité (dans le cas de NeVO). Nous présentons dès maintenant ces trois systèmes.

```
#
# Configuration file for host pooh
#
# $ Id : poo.dsl 1.6 2000/05/30 12 :26 :58 dds Exp $
#
HZ = 100            # "Floating point" value divisor
bigend = FALSE      # Set to TRUE for big endian (e.g. Sun), FALSE for
                    # little endian (e.g. VAX, Intel x86)
map = TRUE          # Set to TRUE to map uid/tty numbers to names
EPSILON = 150       # New maxima difference threshold (%)
report = TRUE       # Set to TRUE to report new/updated entries
unlink = FALSE      # Set to TRUE to start fresh
# Reporting procedure
output = '| /usr/bin/tee /dev/console | /bin/mail root'
# Databases and parameters to check
dbcheck(tty, minbmin, maxbmin, maxio, maxcount)     # Terminals
dbcheck(comm, ALL)                                  # Commands
dbcheck(uid, ALL)                                   # Users
dbcheck(uidtty, maxcount)                           # Users on a terminal
dbcheck(uidcomm, minbmin, maxbmin, maxutime,        # Users of a command
        maxstime, maxmem, maxrw, maxcount, maxasu)
# Map users and terminals into groups
usermap(caduser, john, marry, jill)
usermap(admin, root, bin, uucp, mail, news)
termmap(room202, tty31, tty32, tty33, tty34, tty35)
termmap(ptys, ttyp01, ttyp02, ttyp03, ttyp04, ttyp05, ttyp06)
```

FIG. 1.1 – Exemple de fichier de configuration de Panoptis.

1.1 Panoptis

Panoptis [52] se distingue des autres systèmes de détection d'intrusion présentés dans ce chapitre de par le fait qu'il s'agit du seul système basé sur une détection d'anomalies que nous avons choisi de présenter. Comme nos travaux se situent plutôt dans le cadre de reconnaissance de scénarios, nous avons mis le focus sur ces types de systèmes, mais nous avons tout de même jugé utile, par souci de complétude, d'inclure au moins un détecteur d'anomalies.

Comme nous l'avons déjà dit, Panoptis est configurable à l'aide d'un langage spécifique au domaine. Ce domaine est celui des processus d'un système Unix, et Panoptis prend donc en entrée des fichiers d'audits de processus. Les auteurs affirment que le langage a quand même été conçu de façon assez générale pour pouvoir fonctionner sur n'importe quel système fournissant des fichiers d'audits de processus, dont Windows NT.

Panoptis maintient des tables contenant les profils d'activités pour différents utilisateurs, terminaux, processus, ou groupages et/ou couplage de ces entités. Par exemple, il peut maintenir une table concernant l'utilisation d'un processus donné, pour trois utilisateurs spécifiques utilisant un même terminal. Les informations maintenues pour

```
#
# Panoptis crontab file for host pooh
#
# The format of this file is :
# Hour Minute Day-of-month Month Day-of-week Command
*    5,25,45 * * *   panoptis pooh-quick.cfg pooh.20min    20m
8-18 05      * * *   panoptis pooh-hour.cfg  pooh.workhour 1h
19-7 05      * * *   panoptis pooh-hour.cfg  pooh.late     1h
4    50      * * 1-5 panoptis pooh-day.cfg   pooh.workday  24h
4    50      * * 6,0 panoptis pooh-day.cfg   pooh.weekend  24h
2    20      * * 0   panoptis pooh-full.cfg  pooh.weekly   7d
                          /usr/adm/pacct? /usr/adm/pacct
```

FIG. 1.2 – Exemple de fichier de planification de tâches de Panoptis.

chacune des entités surveillées sont paramétrables à l'aide d'un fichier de configuration. Un exemple de fichier de configuration utilisé par Panoptis se trouve à la figure 1.1. Ce fichier se divise en quatre parties. En premier lieu, on trouve un ensemble de définitions de variables qui servent à définir le fonctionnement général du programme. Ensuite, on indique la methode de rapport à utiliser. La troisième partie est celle où l'on spécifie ce que l'on veut surveiller. Pour chacune des cinq tables `tty`, `comm`, `uid`, `uidtty` et `uidcomm`, on spécifie les champs que l'on veut surveiller. Ces champs sont prédéfinis et font partie de la définition du langage. Le mot-clé `ALL` signifie que l'on veut surveiller tous les champs. Finalement, on trouve les options de groupage qui permettent de regrouper certains utilisateurs ou terminaux en un seul. Par exemple, à la figure 1.1, les utilisateurs `john`, `marry` et `jill` sont regroupés sous l'utilisateur abstrait `caduser`.

Une fois que ces options sont choisies, il reste à configurer la fréquence à laquelle les fichiers d'audits sont lus par le système. Panoptis peut soit les lire en continu, soit être exécuté à des intervalles prédéfinis. Un exemple de fichier de planification de tâches se trouve à la figure 1.2. On voit que des intervalles différents, de même que des fichiers de configuartion différents, peuvent être utilisés selon les périodes de la journée ou de la semaine. Finalement, à chaque fois que Panoptis est exécuté sur un fichier d'audit, celui-ci compare les valeurs calculées à celles se trouvant dans les tables et rapporte les entrées qui présentent des différences significatives.

En résumé, Panoptis constitue un cas typique de système de détection d'intrusion effectuant de la détection d'anomalies. Il est configurable à l'aide d'un langage très simple et spécifique à l'analyse de fichiers d'audits de processus sur un système Unix. Un tel outil fait partie des composantes essentielles à une architecture de sécurité car il permet d'avoir une vision résumée de fichiers d'audits qui, dû à leur grande taille, seraient inutilisables sinon. Comme la plupart des systèmes effectuant de la détection d'anomalies, Panoptis est sujet à un grand nombre de faux positifs et sa nature adap-

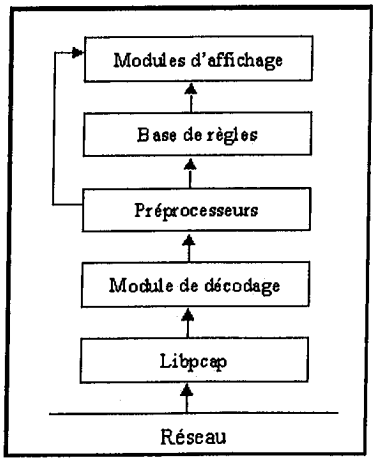

FIG. 1.3 – Structure de Snort.

tative le rend vulnérable aux utilisateurs qui modifieraient tranquillement leur profil afin de ne pas se faire remarquer. Cependant, son mode de fonctionnement générique le rend capable de détecter des attaques qu'un système fonctionnant à base de signatures aurait pu laisser passer.

1.2 Snort

Le système de détection d'intrusions Snort [19], originellement conçu pour être un renifleur (*sniffer*) plus évolué que tcpdump [24], s'est avéré petit à petit très efficace comme outil de détection d'intrusions. En effet, il dispose d'un langage de signatures permettant de décrire avec une grande précision les caractéristiques des paquets offensifs circulant sur le réseau.[1] L'hypothèse derrière le choix de Snort comme système de détection d'intrusion est donc que les attaques sont repérables à l'aide d'un seul paquet. Bien que les efforts investis par la communauté pour développer des langages de signatures permettant de tenir compte de plusieurs paquets - voir tout le reste du présent rapport - incitent à penser qu'un langage de signatures concernant un seul paquet n'est pas suffisamment expressif, la pratique démontre qu'il est tout de même possible de détecter un grand nombre d'attaques sur la base d'un tel langage (la base de signatures de Snort contient, à ce jour, plus de 2000 attaques).

La structure logicielle de Snort se trouve à la figure 1.3. Utilisant, tout comme

[1] Une autre approche consiste à détecter les paquets résultants de l'attaque. Lorsqu'il est possible de définir de tels paquets, le nombre de faux positifs est alors diminué.

tcpdump, la librairie de capture de paquets LibPcap [25], Snort comporte un module de décodage de paquets permettant de décoder un grand nombre de protocoles de différents niveaux. Au dessus de ce module de décodage se situe un ensemble de préprocesseurs, dont la fonction originale était de régulariser le format des paquets afin de faciliter la tâche de la base de règles.

Par exemple, si une politique de sécurité spécifie que personne ne doit se connecter au service Telnet en utilisant l'utilisateur `root`, on voudrait pouvoir spécifier une règle qui surveille les paquets contenant la chaîne de caractères `user : root`. Cependant, il est possible d'échapper à cette règle en utilisant le caractère de retour en arrière <BS>. En tapant `user : roo<BS>ot`, un utilisateur peut alors se connecter sous le compte `root` sans se faire repérer. Des astuces similaires s'appliquent au trafic HTTP. Pour ces raisons, Snort comporte des préprocesseurs responsables de régulariser le trafic Telnet et HTTP. Il est cependant possible d'échapper au système de détection d'intrusions autrement qu'en utilisant des astuces de formattage au niveau application. Une astuce bien connue consiste à fragmenter le paquet offensif en plusieurs petits paquets, de façon à ce qu'aucun des paquets résultants ne soit alarmant. Une autre façon de faire est d'envoyer les paquets dans un ordre ne correspondant pas à celui dans lequel ils seront interprétés par la machine qui les recevra. Les préprocesseurs frag2 et stream4 ont donc été ajoutés par l'équipe de programmeurs de Snort afin de remettre ensemble les paquets fragmentés et de présenter les paquets au système de détection d'intrusions dans le même ordre que celui dans lequel le système les recevant les interprètera. Il s'agit encore une fois d'un travail de formattage visant à éviter certaines attaques d'évasion.

En plus de cette fonctionnalité, les préprocesseurs sont aussi utilisés à deux autres fins : détecter les scénarios d'attaques utilisant plusieurs paquets, ainsi que permettre l'acquisiton passive d'information. Un exemple simple de catégorie d'attaques nécessitant la détection de plusieurs paquets est le balayage (*scan*). Qu'il s'agisse d'un balayage TCP, UDP, ICMP ou autre, les attaques de balayage ont pour objectif de permettre à l'attaquant d'acquérir de l'information sur le réseau ou sur la machine victime afin de mieux préparer une attaque. Les balayages peuvent s'effectuer en utilisant des fontionnalités tout à fait normales de la pile TCP/IP, et la seule façon de les détecter consiste généralement en la détection de l'utilisation abusive de ces fontionnalités. Certains préprocesseurs (par exemple portscan2) ont donc été ajoutés afin de compter, par exemple, le nombre de connexions TCP demandées par un hôte distant dans une fenêtre de temps donnée.

Effectuer un minimum d'acquisition est plus que souhaitable pour un système de détection d'intrusions : il s'agit d'une fonctionnalité vitale. Un exemple de préprocesseur effectuant de l'acquisition passive d'information est le préprocesseur flow, dont la

```
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg :"SCAN nmap TCP" ;
stateless ; flags :A,12 ; ack :0 ; reference :arachnids,28 ;
classtype :attempted-recon ; sid :628 ; rev :3 ;)
```

FIG. 1.4 – Exemple de signature Snort.

```
alert tcp any any <> 192.168.1.0/24 80 (content : "sex" ;
msg : "Not for children !" ; react : block, msg ;)
```

FIG. 1.5 – Exemple d'utilisation du module de réaction.

tâche est de tenir à jour une table des sessions TCP actives. Avant l'arrivée de flow, il était possible d'exploiter le fait que Snort fonctionnait paquet par paquet pour générer automatiquement des paquets correspondant aux signatures d'attaque [18], engorgeant ainsi le système de détection d'intrusions de faux-positifs.

Au dessus de tous ces préprocesseurs, se trouvent les modules de détection. Ces derniers sont responsables de reconnaître les profils de paquets offensifs. La description des paquets à reconnaître se trouve dans un fichier texte qui est lu à l'amorçage du programme. Le langage utilisé pour décrire ces paquets est relativement simple, ce qui explique en partie la popularité et la croissance rapide du nombre d'attaques que Snort est maintenant capable de reconnaître. Typiquement, une signature Snort est une suite de mots-clé suivis de un ou plusieurs (parfois aucun) arguments. Certains mot-clés ne concernent pas la détection, mais sont simplement ajoutés afin de documenter l'attaque. Une fois les paquets reconnus, une série de modules d'affichage et d'archivage sont responsables de l'interaction avec l'usager. Lorsque les attaques sont identifiées, Snort peut, entre autres, soit afficher un message à l'écran, ajouter une ligne dans un fichier texte, une entrée dans une base de données, envoyer un message via le protocole SNMP, etc.

À la figure 1.4, on trouve un exemple de signature Snort. L'interprétation de cette signature se lit comme suit : si un paquet TCP provenant de l'extérieur (`$EXTERNAL_NET`) pénètre dans notre réseau (`$HOME_NET`), peu importe les ports (`any`), et que ce paquet a le drapeau ACK activé, de même que les deux bits réservés (`flags :A,12`), et que le numéro d'acquiescement est 0 (`ack :0`), peu importe l'état de la session (`stateless`), il faut alors signaler un balayage TCP fait avec nmap [17] (`msg :"SCAN nmap TCP"`). Le reste de la signature constitue la documentation de l'attaque.

Cette revue de Snort serait incomplète si aucune allusion n'était faite à la nature

particulière des modules de détection react et flowbits. Le module react fait de Snort non-plus un simple système de détection d'intrusions, mais un système de prévention d'intrusions. Il permet de réagir en interrompant les sessions TCP jugées offensives ou allant à l'encontre de certaines politiques. Il permet, de plus, d'envoyer un message d'avertissement à l'usager expliquant pourquoi la communication a été interrompue. À la figure 1.5, on voit comment ce module peut être utilisé pour empêcher les enfants d'aller sur des sites pornographiques. Lorsqu'un usager tente de consulter un site web contenant le mot **sex**, la session TCP correspondante est interrompue et le message **not for children** est affiché dans son navigateur. Cet exemple est tiré de [19].

Le module flowbits, quant à lui, a été mis au point afin de répondre au besoin de plus en plus grandissant de permettre à Snort de tenir compte de l'état des sessions au niveau application. Il permet de définir des variables booléennes (drapeaux) servant à suivre l'état de la session en changeant la valeur de ces variables (`set`, `unset`, `reset`, `toggle`) dans certaines règles, et en les lisant dans d'autres (`isset`, `isnotset`). Comme ce module a été mis au point pour suivre l'état des sessions au dessus de TCP, ces variables sont instanciées et initialisées à chaque nouvelle session TCP. Il à noter que comme les règles servant à mettre à jour les valeurs des drapeaux ne doivent pas générer d'alarmes, celles-ci contiennent généralement le mot-clé `noalert`, signifiant que même si la règle est activée, aucune alerte ne doit être générée.

En résumé, Snort est un système de détection d'intrusions ouvert jouissant d'une grande popularité. Il fonctionne au niveau réseau et sa base de signatures contient la description d'un nombre considérable d'attaques. Il comporte plusieurs préprocesseurs qui permettent de faciliter le travail des modules de détection ainsi que de protéger le système d'évasions communes et d'attaques d'inondation de faux-positifs. Les fonctionnalités offertes par ces modules devraient faire partie intégrante de tout bon système de détection d'intrusions. Les modules de détection, situés au-dessus des préprocesseurs, permettent à l'utilisateur de définir ses propres règles dans un langage simple. Certains moyens ont été mis en oeuvre pour compenser les manques les plus importants reliés à la nature *un paquet-une signature* de ce langage. Cependant, à mesure que ces moyens sont mis en place, il devient de plus en plus évident qu'un langage de spécification idéal devrait pouvoir tenir compte de plusieurs paquets. Le reste de ce rapport est donc consacré aux langages permettant d'exprimer des signatures s'étalant sur plusieurs paquets (ou événements).

1.3 NeVO

NeVO [14] n'est pas un système de détection d'intrusions, mais un identificateur passif de failles de sécurité. La raison pour laquelle nous le mentionnons ici est que la connaissance des failles de sécurité du réseau fait partie des moyens étant souvent mentionnés pour réduire le nombre de faux positifs générés par un système de détection d'intrusions [20, 39]. Dans [20], on trouve une courte étude des différentes façons dont les identificateurs de failles de sécurité pourraient ou devraient, selon les besoins, coopérer avec les systèmes de détection d'intrusion. Par exemple, la connaissance des failles de sécurité peut être utilisée pour activer ou désactiver certaines règles du système de détection d'intrusions, diminuant ainsi à la fois le nombre de faux positifs et la demande faite au processeur de la machine sur laquelle le système s'exécute. D'autres pensent plutôt que toutes les attaques devraient continuer à être surveillées, sauf que l'outil de visualisation utilisé pour consulter le journal d'attaques devrait permettre d'identifier celles dont le succès est le plus vraisemblable étant donné les vulnérabilités identifiées. Réciproquement, un autre modèle de coopération envisageable serait de lancer l'identificateur de failles seulement au moment où les attaques surviennent, allégeant ainsi le travail de ce dernier plutôt que celui du système de détection d'intrusions. Finalement, on ne doit pas oublier, lors de l'étude de ces différents modèles, que les identificateurs de failles sont autant sujets aux faux positifs et aux faux négatifs que les systèmes de détection d'intrusion.

NeVO, mis au point par la compagnie Tenable, est un identificateur passif de failles de sécurité, ce qui signifie qu'il effectue son travail sans avoir à injecter de trafic sur le réseau. Il se présente comme un complément ou une alternative à Nessus [3, 4, 5], qui est un identificateur actif de failles de sécurité. Bien que l'identification active puisse en général fournir des rapports plus exhaustifs que l'identification passive, elle peut souvent s'avérer très dérangeante pour le réseau en cours d'audit, voire même mener à son instabilité. De plus, l'identification passive permet d'avoir accès à une information continuellement mise à jour. Finalement, l'identification passive permet non-seulement de trouver les failles sur les serveurs, mais aussi sur les clients. Pour toutes ces raisons, l'identification passive s'impose comme un complément essentiel à l'identification active.

À première vue, un identificateur passif de failles de sécurité ne devrait pas fonctionner différemment d'un système de détection d'intrusions. D'un point de vue abstrait, on tente d'identifier un comportement caractéristique de l'information à laquelle on s'intéresse. Les auteurs de [14] citent cependant deux différences notables entre le mode de fonctionnement d'un identificateur de failles et celui d'un système de détection d'intrusions : la tolérance à l'échantillonnage et le modèle d'accès à l'information.

Alors que le fait de n'être pas capable de traiter en temps réel tout le trafic circulant sur le réseau peut être considéré pour un système de détection d'intrusions comme une faiblesse majeure, cette particularité est beaucoup plus tolérable pour un identificateur passif de failles de sécurité. L'hypothèse permettant de dire qu'un identificateur de failles peut fonctionner aussi bien sur une base échantillonnale qu'en observant la totalité du trafic est que le comportement caractéristique de la faille sera observable pratiquement à chaque fois que le système défaillant entrera en communication. Si on n'identifie pas la faille maintenant, on l'identifiera bien plus tard.

L'information acquise par un identificateur passif de failles de sécurité ne doit pas, contrairement à celle acquise par un système de détection d'intrusions, être archivée à chaque fois qu'elle est détectée. Au contraire, l'identificateur doit plutôt tenir à jour un modèle du réseau et le fournir sur demande. Cette demande sera typiquement faite au moment de la consultation du journal d'attaques du système de détection d'intrusions. L'information n'est donc pas poussée, mais conservée jusqu'à ce qu'elle soit demandée.

Ces deux différences concernant l'architecture logicielle d'un identificateur passif de failles ne justifient cependant pas le fait que le langage utilisé pour décrire les façons de détecter les failles soit différent de celui utilisé par un système de détection d'intrusions, et c'est pourquoi nous nous attarderons maintenant au langage de NeVO. Le langage de NeVO est un langage fortement dédié à l'acquisition passive d'information sur un réseau informatique. De plus, l'acquisition d'information faite à l'aide de ce langage se limite au niveau application. Cela signifie que l'inspection faite sur les paquets se limite à la partie données de ceux-ci, et non aux différents champs des différents protocoles ni à l'interaction de ceux-ci. NeVO incorpore cependant un module permettant de détecter passivement les systèmes d'exploitation des machines communiquant sur le réseau en inspectant les entêtes des paquets SYN, mais ce module n'utilise pas le langage de NeVO.

Une signature NeVO concerne généralement un seul paquet. On spécifie le contenu recherché à l'aide du mot-clé `match`. Cependant, il est possible de spécifier, à l'aide du mot-clé `pmatch`, le contenu du paquet précédent dans la même session. Par défaut, le contenu est recherché en mode texte. Il est cependant possible de rechercher un contenu binaire à l'aide du mot-clé `bmatch`. Le mot-clé `regex` permet de rechercher des expressions régulières. Le mot-clé `dependency` permet de spécifier qu'une signature ne doit être considérée, pour un flux TCP donné, que si une autre signature a été activée. Le mot-clé `time-dependency` permet de limiter le délai d'attente entre l'activation des deux signatures. Le mot-clé `description` permet de spécifier le message a afficher et le mot-clé `nooutput` signifie qu'il n'y a pas de message à afficher. On trouve à la figure 1.6 un exemple de signature NeVO (tiré de [14]) responsable d'identifier les serveurs FTP

```
id=700018
nooutput
hs_sport=21
name=Anonymous FTP (login : ftp)
pmatch=^USER ftp
match=^331
NEXT #------------------------------
id=700019
dependency=700018
timed-dependency=5
hs_sport=21
name=Anonymous FTP enabled
description=The remote FTP server has anonymous access enabled.
risk=LOW
pmatch=^PASS
match=^230
```

FIG. 1.6 – Exemple de signature NeVO.

ayant un compte *anonymous* actif. Cette signature se divise en deux sous-signatures. La première est responsable d'identifier les paquets contenant une demande de connection pour l'utilisateur `ftp` suivie d'une acceptation du nom d'utilisateur (code 331). La seconde, activée seulement dans les 5 secondes suivant l'activation de la première, est responsable d'identifier l'envoi d'un mot de passe suivi de l'acceptation de ce mot de passe (code 230).

En résumé, NeVO est un exemple très intéressant d'identificateur passif de failles de sécurité. Même s'il a été développé à des fins commerciales (habituellement, la documentation disponible sur les outils commerciaux se limite aux aspects externes), on peut trouver une documentation riche et précise sur son fonctionnement interne. À première vue, son langage dédié à l'identification de simples paquets au niveau application pourrait nous porter à croire que le nombre de comportements observables est plutôt limité, mais la possibilité de spécifier une notion de dépendance entre les règles augmente sans aucun doute l'expressivité du langage de façon significative. Cependant, cette façon ad hoc de régler le problème des scénarios complexes ne saurait amener toute l'expressivité désirable. Entre autres, elle ne permet pas de reconnaître le fait qu'un paquet donné n'arrive pas dans un certain délai, ou encore de compter le nombre d'occurences d'un paquet donné. De plus, il est dommage que la sémantique du langage se limite au niveau application. Le fait de pouvoir spécifier des paquets au niveau des différentes couches de protocoles permettrait sans doute de détecter plus de failles et, sans nécessairement de limiter aux failles, d'acquérir encore plus d'information.

Chapitre 2

Langages impératifs

La distinction entre les langages impératifs et les langages déclaratifs, dans la littérature, n'est pas aussi nette que l'on pourrait le croire. Généralement, on se contente de définir informellement un langage déclaratif comme étant un langage permettant de définir *ce que* l'on veut identifier, plutôt que *comment* l'identifier. Les autres langages tombent tous sous la catégorie des langages impératifs. Sur la base de cette définition, le langage de signatures de Snort, par exemple, est clairement déclaratif.

À l'exception des langages présentés dans le présent chapitre, les langages que nous présentons dans ce rapport ont tous la prétention d'être déclaratifs. Plus loin dans ce rapport (page 22), nous proposerons une définition plus formelle permettant de départager les langages déclaratifs des langages impératifs. Pour l'instant, nous présentons deux langages qui, peu importe la définition utilisée, appartiennent certainement à la catégorie des langages impératifs. RUSSEL, que nous présentons dans un premier lieu, a été conçu dans le but d'analyser séquentiellement des fichiers d'audit, alors que Bro, que nous présentons par la suite, a été conçu afin de pouvoir exprimer les signatures dans un langage typé proche de celui utilisé pour exprimer les politiques de sécurité dans un contexte de détection d'intrusion au niveau réseau.

2.1 ASAX

ASAX [21, 22, 40] a été développé dans le but d'analyser des traces d'audit. La syntaxe de RUSSEL, le langage utilisé pour écrire les signatures, se trouve à la table 2.1. L'exécution d'un programme RUSSEL implique l'analyse séquentielle d'un fichier de

R	: :=	rule Q (... ; G; ...); ... ; H; ... ; A
G	: :=	P : T
H	: :=	V : T
A	: :=	V := E | Y (... , E , ...)
		| trigger off M Q (... , E , ...) | begin ... ; A ;... end
		| do ... ; C → A; ... od | if ... ; C → A; ... fi
C	: :=	true | F present | not C | C B C | E S E
E	: :=	L | V | F | P | -E | E O E | X(...,E,...)
B	: :=	and | or
O	: :=	+ | - | * | div | mod
S	: :=	> | < | = | ≠ | ≤ | ≥
M	: :=	for current | for next | at completion
T	: :=	integer | byte | string

A	actions	O	arithmetic operators
B	logical operators	P	formal parameters
C	conditions	Q	rule names
E	expressions	R	rules
F	field names	S	relational operators
G	parameter declarations	T	types
H	variable declarations	V	local variables
L	literals	X	external function names
M	triggering modes	Y	external procedure names

TAB. 2.1 – Syntaxe abstraite de RUSSEL.

```
01.rule Failed_login (maxtimes, duration : integer)
02.# This rule detects a first failed login and triggers off an accounting rule with an
03.# expiration time
04.begin
05.if evt='login' and res='failure' and is_unsecure (terminal)
06.   → Trigger off for next Count_rule1 (maxtimes-1, timestp+duration)
07.fi;
08.Trigger off for next Failed_login (maxtimes, duration)
09.end
10.
11.rule Count_rule1 (countdown, expiration : integer)
12.# This rule counts the subsequent failed logins,
13.# it remains active until its expiration time or until the countdown becomes 0
14.if evt='login' and res='failure'
15.   and is_unsecure(terminal) and timestp < expiration
16.   → if countdown > 1
17.      → Trigger off for next Count_rule1(countdown-1, expiration);
18.      countdown =1
19.      → SendMessage ("too much failed login's")
20.   fi;
21.   timestp ≥ expiration
22.   → Skip;
23.   true
24.   → Trigger off for next Count_rule1 (countdown, expiration)
25.fi
```

FIG. 2.1 – Détection d'une pénétration externe dans ASAX.

trace d'audit. Une trace d'audit est définie comme étant une séquence finie d'enregistrements, qui eux sont définis comme étant une fonction partielle associant des valeurs à un ensemble d'étiquettes, aussi appelées champs. Ces enregistrements sont passés à ASAX dans un format normalisé (*Normalized Audit Data Format*), qui permet de faire abstraction du type d'audit.

La notion centrale de ASAX est celle de contexte d'exécution. Un contexte d'exécution est entre autres caractérisé par trois ensembles de règles : *Cur*, *Nxt* et *Cmp*. L'ensemble *Cur* contient les instances de règles qui doivent être exécutées sur l'enregistrement courant, l'ensemble *Nxt* contient les instances de règles qui doivent être exécutées sur l'enregistrement suivant, et l'ensemble *Cmp* contient l'ensemble des règles qui doivent être exécutées une fois que tous les enrgistrements sont traités. L'utilité de ce dernier ensemble est de permettre d'effectuer des traitements sur des valeurs ayant été accumulées tout au long du traitement de la trace d'audit. Par exemple, il permet de calculer une moyenne et de la sauvegarder à quelque part. Ces trois ensembles doivent être maintenus à jour explicitement par celui qui écrit les règles via le mot-clé **Trigger off**.

Une autre caractéristique d'un contexte d'exécution est l'ensemble des valeurs des différentes variables. Ces variables peuvent être de trois types : locales à une règle, glo-

bales, ou correspondre aux champs d'un enregistrement. Comme l'ordre d'exécution des règles n'est pas spécifié, l'utilisation de variables globales peut amener un certain non-déterminisme. Aussi, pour palier aux problèmes découlant du fait que certains champs peuvent être absents de certains enregistrements, on a ajouté un mot-clé **present**, permettant de tester la présence d'un champs. Lorsqu'un champs est absent, il est tout de même possible de lire sa valeur. La valeur lue est alors celle qui était présente la dernière fois que le champs était présent, et les résultats obtenus sont susceptibles d'être très différents de ceux escomptés.

À la figure 2.1 se trouve un exemple de signature écrit en RUSSEL pour ASAX. Cet exemple a été copié tel quel de [21]. Les variables *evt* et *res* sont des champs qui sont supposés faire partie de chaque enregistrement. Ensemble, les deux règles `Failed_login` et `Count_rule1` servent à surveiller un utilisateur qui échouerait de se connecter trop souvent. On remarque que la règle `Failed_login` se réactive inconditionnellement (ligne 8). La règle `Count_rule1` démontre la sémantique particulière d'un bloc **if**...**fi**. Ces blocs renferment une série de conditions, chacune suivie d'un bloc d'actions. Dès qu'une condition est vérifiée, le bloc correspondant est exécuté et le bloc **if**...**fi** est terminé. L'instrcution **skip** est l'instruction qui ne fait rien. Donc, si le délai est écoulé (ligne 21), la règle n'est pas réactivée. Sinon, la condition **true** est vérifiée et la règle est réactivée pour le prochain enregistrement.

En résumé, RUSSEL est un langage de programmation impératif comportant, en plus des notions les plus communes, celles de contexte d'exécution et de variables d'enregistrement. Ces notions permettent d'automatiser certaines tâches communes au traitement séquentiel de fichier d'audit telles que la mise à jour de la base de règles et la mise à jour des valeurs des champs des enregistrements. Cependant, il n'est pas du tout clair que l'approche consistant à développer un nouveau langage ait été ici la meilleure à adopter. Les notions particulières au langage RUSSEL auraient très bien pu être implantées sans développer un nouveau langage. Par exemple, on aurait pu envisager une approche orientée-objet dans un langage déjà connu de la communauté (tel que C++), et arriver à des résultats semblables sinon meilleurs du point de vue de la facilité de maintenance et de la courbe d'apprentissage.

2.2 BRO

Bro [42] est un système de détection d'intrusions conçu spécifiquement en vue de détecter les attaques survenant au niveau réseau. Son architecture logicielle en couches, que l'on peut voir à la figure 2.2, est comparable à celle de Snort. À la base, se trouve la

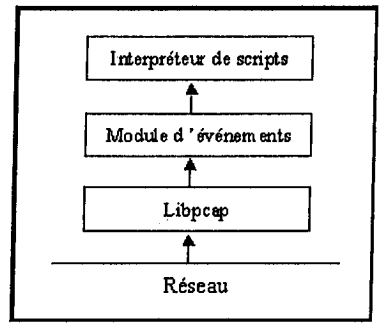

FIG. 2.2 – Structure de Bro.

même librairie de capture de paquets (libpcap). Le module d'événements fait un travail relativement semblable à celui auquel les préprocesseurs de Snort, conjointement au module de décodage, étaient initialement destinés : faire un premier travail de formattage sur les paquets afin de simplifier la tâche de l'interpréteur de scripts de politiques de sécurité. Ce dernier est comparable au module de détection. Il n'y a pas, pour Bro, d'équivalent des modules d'affichage. Tout est archivé dans des fichiers en format texte.

Malgré le fait que les deux architectures puissent se ressembler au premier abord, il y a tout de même deux différences majeures entre les deux systèmes : l'abstraction créée par le module d'événements et la nature impérative du langage de signatures. Alors que les objets fournis par les préprocesseurs de Snort au module de détection sont des paquets, ceux fournis par le module d'événements de Bro à l'interpréteur de scripts sont d'un type plus abstrait : celui d'événement. Ceci implique qu'un processus de filtrage est fait par le module d'événements, et que seulement les informations jugées pertinentes sont passées à l'interpréteur de scripts. Un des objectifs de cette étape de filtrage est de permettre au système de fonctionner en temps réel, malgré la grande affluence de trafic pouvant circuler sur le réseau. Aussi, les politiques de sécurité vérifiées avec Bro sont normalement plus haut niveau que celles vérifiées par Snort, voire même la plupart des autres systèmes de détection d'intrusion. En fait, il s'agit d'un des objectifs poursuivis par les auteurs : permettre une séparation claire entre les mécanismes et les politiques de sécurité. Les mécanismes, dans ce cas, sont les différents moyens de générer un événement donné, alors que les politiques de sécurité sont les actions à prendre lorsque les événements sont identifiés. Les mécanismes sont donc gérés au niveau du moteur d'événements, programmé en C++, alors que les politiques de sécurité sont mises en oeuvre par l'interpréteur de scripts. Ces dernières sont écrites dans un langage de signatures propre à Bro.

Alors que le langage de signatures de Snort est un langage déclaratif décrivant certaines caractéristiques bien ciblées des paquets circulant sur le réseau, celui de Bro est

```
01.global hot_names = { "root", "lp", "uucp" };
02.global finger_log = open("finger.log");
03.event finger_request(c :connection, request : string, full : bool)
04.{
05.   if ( byte_len(request) > 80 ) {
06.      request = fmt("%s...", sub_bytes(request, 1, 80));
07.      ++c$hot;
08.   }
09.   if ( request in hot_names )
10.      ++c$hot;
11.   local req = request == ""? "ANY" : fmt("\"%s\"", request);
12.   if ( c$addl != "" )
13.      # This is an additional request.
14.      req = fmt("(%s)", req);
15.   if ( full )
16.      req = fmt("%s (/W)", req);
17.   local msg = fmt("%s > %s %s", c$id$orig_h, c$id$resp_h, req);
18.   if ( c$hot > 0 )
19.      log fmt("finger : %s", msg);
20.   print finger_log, fmt("%.6f %s", c$start_time, msg);
21.   c$addl = c$addl == ""? req : fmt("*%s, %s", c$addl, req);
22.}
```

FIG. 2.3 – Exemple de script Bro.

un langage impératif, quasi aussi expressif que le langage C (sa syntaxe en est d'ailleurs en plusieurs points très semblable), avec déclaration de variables (locales ou globales), expressions composées, inférence et vérification de types, etc. Un des objectifs visé par le développement de ce langage est de permettre, au moment de la compilation, d'effectuer un maximum de vérification de types, de façon à ce qu'au moment de l'exécution, le type des variables référenciées soit cohérent. Pour ce faire, le langage de Bro dispose de plusieurs types de variables propres à la réseautique : port, adresse IP, nom réseau, etc. Le langage dispose aussi de structures de données abstraites : ensemble, table d'associations, enregistrement; permettant d'exprimer naturellement des vérifications dans un langage proche des politiques de sécurité : ***si** il y a une connexion Telnet sur un des serveurs d'impression, **alors**...*. Il est à noter que bien que le langage de Bro soit impératif, celui-ci ne dispose pas de construction syntaxique itérative. Cette caractéristique a été voulue par les concepteurs afin de minimiser le temps de traitement d'un événement. Cependant, comme la récursivité est tout de même permise, l'itération est toujours possible. Les concepteurs n'ont toutefois pas relevé de cas ou son utilisation se faisait ressentir.

À la figure 2.3, on peut voir un exemple de script de politique de sécurité écrit dans le langage de Bro. Cet exemple, tiré de [42], montre comment on doit réagir à une requête *finger*. L'opérateur $ sert à accéder aux différents champs des structures. L'opérateur in (ligne 9) sert à vérifier si un élément fait partie d'un ensemble. Le script vérifie en premier lieu si la requête est excessivement longue (ligne 5). Le cas advenant, celle-ci

est tronquée et le champs `hot` est incrémenté. Ensuite, il vérifie si la requête concerne un nom sensible (ligne 9). Si tel est le cas, le champs `hot` est encore incrémenté. La variable locale `req` est ensuite initialisée (lignes 11 à 16) et, si le champs `hot` a été incrémenté (ligne 18), une notification en temps réel est effectuée (ligne 19). Dans tous les cas, la requête est archivée (ligne 20). Finalement, la requête est enregistrée dans le champs `addl` de la connexion (ligne 21). Advenant le cas où une autre requête a eu lieu dans le contexte de la même connexion, ce qui peut être associé à un comportement malicieux, elle est ajoutée à celle s'y trouvant déjà et un astérisque est ajouté pour attirer l'attention lors de l'inspection du fichier d'audit.

L'algorithme de vérification en temps-réel de Bro fonctionne en une seule passe et suivant un seul fil d'exécution (*thread*). Lorsqu'un paquet est capturé sur le réseau par la librairie libpcap, il est passé au moteur d'événements. Celui-ci peut alors générer de zéro à plusieurs événements pour un seul paquet (réciproquement, un type d'événement donné peut être généré par plusieurs types de paquets différents). Ces événements sont alors placés dans une file d'attente et, par la suite, traités par l'interpréteur de scripts de politiques de sécurité. Pour un événement donné, il peut y avoir plusieurs traitements à faire. Ceux-ci sont effectués dans l'ordre où ils apparaissent dans le fichier de script. Il est à noter que ces traitements peuvent, en plus de générer des alarmes, créer d'autres événements qui iront se placer à la fin de la file et devront être traités avant de retourner au moteur d'événements et de passer au paquet suivant.

Les travaux entourant Bro sont grandement motivés par la pratique et l'article [42] traite de plusieurs problèmes concrets devant être pris en compte lors de la conception d'un système de détection d'intrusion en temps réel. L'auteur parle entre autres des différentes façons d'attaquer le système lui-même et des moyens de s'en défendre, des problèmes reliés à la surveillance de réseaux haut-débit (principalement en ce qui concerne la perte de paquets), des différentes façons de gérer les chronomètres, des avantages et des inconvénients de disposer d'un langage compilé, des problèmes reliés au redémarrage du système (souvent nécessaire pour faire un nettoyage de la mémoire), etc. Quiconque s'engage dans des travaux reliés à la détection d'intrusion en temps réel devrait au moins prendre quelques minutes pour jeter un oeil à cet article.

En résumé, Bro est un système de détection d'intrusions fonctionnant au niveau réseau dont la conception a grandement été influencée par des problèmes pratiques. Il vient avec un langage impératif interprété qui permet d'exprimer des scripts vérifiant des politiques de sécurité à un plus haut niveau que la plupart des autres systèmes de détection d'intrusion. Son architecture en couches, semblable à celle de Snort, a été conçu avec un souci d'extensibilité et permet d'ajouter des modules d'événements et de définir nos propres politiques de sécurité assez facilement. Le langage utilisé pour

exprimer les politiques de sécurité reste cependant impératif, ce qui implique que l'on indique plutôt *comment* vérifier les politiques, que les politiques elle-mêmes. Il faut tout de même donner à Bro le mérite d'avoir eu l'idée de séparer les mécanismes des politiques au niveau même de l'architecture du logiciel et d'avoir prévu deux modules différents pour gérer ces deux aspects.

Chapitre 3

Systèmes de transitions

Les systèmes de transition jouissent d'une grande popularité en informatique lorsque vient le temps de décrire des programmes ou des processus de façon abstraite. Les objectifs visés par ces descriptions varient de la simple communication entre les différents intervenants dans le processus de développement à la vérification automatique de propriétés de programmes. Les systèmes de détection d'intrusion que nous allons présenter dans ce chapitre utilisent différents formalismes basés sur des systèmes de transition pour représenter les attaques. STAT, dans un premier lieu, utilise les automates généralisés, qui diffèrent des automates traditionnels par la présence de variables dans les états permettant de relier les différents symboles de l'alphabet (ou événements) entre eux. IDIOT, dans un second lieu, utilise les réseaux de Petri colorés, qui diffèrent des réseaux de Petri conventionnels de la même façon. L'avantage des réseaux de Petri sur les automates est de permettre une représentation du synchronisme et du parallélisme de façon plus naturelle. En troisième et dernier lieu, nous présenterons BSML, qui se base sur les expressions régulières *pour les événements*, qui diffèrent elles aussi des expressions régulières par la présence de variables du premier ordre, ainsi que de contraintes de temps. Même si les expressions régulières sont connues pour avoir le même pouvoir de représentation que les automates, BSML n'a pas exactement la même expressivité que STATL. Ceci est dû au fait que STATL, comme nous le verrons, utilise plusieurs types de transitions pour spécifier l'algorithme de vérification relié à l'automate, alors que les expressions de BSML sont traduites en automates généralisés avec seulement des transitions classiques.

Chacun des systèmes que nous présentons dans ce chapitre a été conçu dans l'idée de permettre de représenter les signatures d'attaque d'une façon déclarative. Les auteurs de ces travaux, surtout en ce qui concerne STATL et IDIOT, considèrent que les systèmes de transition constituent un moyen de représenter *ce qui* doit être détecté plutôt que

comment le détecter. Le dictionnaire en ligne Foldoc[1] propose, pour le terme *langage déclaratif*, la définition suivante :

Définition 3.1 (Langage déclaratif) *Any relational language or functional language. These kinds of programming language describe relationships between variables in terms of functions or inference rules, and the language executor (interpreter or compiler) applies some fixed algorithm to these relations to produce a result.*

Declarative languages contrast with imperative languages which specify explicit manipulation of the computer's internal state; or procedural languages which specify an explicit sequence of steps to follow.

The most common examples of declarative languages are logic programming languages such as Prolog and functional languages like Haskell.

Tous les langages étudiés dans le reste de ce rapport disposent d'un algorithme fixe de vérification, comme mentionné dans le premier paragraphe de la définition. Cependant, comme nous le verrons, plusieurs sont aussi pourvus de constructions syntaxiques qui permettent de modifier explicitement le contenu de la mémoire. En se référant au second paragraphe de la définition, on réalise alors que l'on se rapproche de la frontière entre langage déclaratif et langage impératif.

Maintenant que nous avons une définition plus claire de la différence entre langage déclaratif et impératif, nous sommes prêts à étudier les systèmes de détection d'intrusions basés sur les systèmes de transition, qui sont à la frontière entre les deux types de langages.

3.1 STAT

STAT (*State Transition Analysis Technique*) [16] utilise une approche qui se veut déclarative. Le formalisme utilisé pour spécifier les signatures se base sur les automates généralisés. STAT fournit un langage général, STATL, permettant de spécifier des automates. La sémantique attribuée à ces systèmes dépend de l'extension utilisée. Les différentes extensions permettent de faire différents types de détection d'intrusion. Par exemple, les extensions USTAT [23] et WinSTAT permettent de spécifier des signatures au niveau hôte pour les systèmes d'exploitation Unix et Windows, alors que l'extension NetSTAT [53] permet de spécifier des signatures au niveau réseau.

[1] `www.foldoc.org`

Les systèmes de transition définis dans STATL sont constitués d'états et de transitions. Les transitions, qui peuvent être activées par les actions effectuées sur le système, sont de trois types : *consuming*, *non-consuming*, et *unwinding*. Les transitions de type *consuming* correspondent exactement aux transitions régulières des automates finis : lorsque l'action correspondant à l'étiquette de la transition est effectuée, le système change d'état. Les transitions *non-consuming* créent une copie du système avant de changer d'état. Les transitions *consuming* permettent donc de spécifier des propriétés de *la prochaine action telle que*, alors les transitions *non-consuming* permettent de spécifier *l'existence* d'une *prochaine action telle que*. Finalement, les actions *unwinding* permettent d'éliminer des systèmes. Plus spécifiquement, ils permettent d'éliminer toutes les copies ayant été faites du système depuis qu'il a traversé un état donné. Ceci permet de représenter le fait qu'une action peut tout annuler.

À la figure 3.1, on peut voir un exemple, tiré de [16], de scénario exprimable à l'aide du langage STATL. Ce scénario permet de détecter une session TCP à demi ouverte. Une session TCP est dite à demi ouverte si le serveur a accepté une demande de connexion (le SYN et le SYN-ACK ont eu lieu) et est en attente de la confirmation (le dernier ACK) de la part de l'hôte ayant demandé la connexion. Si, pour un serveur donné, trop de connexions sont simultanément à demi ouvertes, celui-ci peut cesser d'accepter de nouvelles connexions. Il est possible d'exploiter ce fait pour effectuer une attaque de déni de service, connue sous le nom de *SYN-Flood*. Le scénario de la figure 3.1 comporte donc trois états : l'état initial (`s0`), l'état *en attente de confirmation* (`s1`), et l'état final `s2`. Le scénario est paramétré par une variable, `timeout`, indiquant combien de temps on doit attendre la confirmation. La première transition du scénario, `SYN`, est activée lorsqu'un paquet de type SYN (ayant le drapeau SYN activé et le drapeau ACK désactivé) est identifié. Cette transition est de type *non-consuming*, signifiant que lorsqu'elle est activée, une copie du scénario est effectuée avant de passer à l'état `s1`. Aussi, lors de l'activation de cette transition, les variables locales du scénario servant à mémoriser les adresses IP et les ports TCP de la victime et de l'attaquant sont initialisées selon les valeurs contenues dans le paquet. On voit bien, avec la transition `SYN`, les deux parties constituantes d'une transition : la condition d'activation et les actions à poser lors de l'activation. En arrivant dans l'état `s1`, le chronomètre `t0` est démarré avec comme délai la valeur de la variable `timeout` fournie en paramètre. Si, avant l'écoulement du délai, un paquet ACK ou un paquet RST sont vus pour les adresses IP et les ports TCP correspondant au SYN (transitions `ACK` et `RST`), tout est annulé et le scénario retourne à l'état original car ces transitions sont de type *unwinding*. Si, au contraire, aucun de ces paquets n'est vu avant que le délai ne s'écoule, la transition `Timed_out` est activée et le scénario passe (sans duplication, puisque la transition est de type `consuming`) à l'état `s2`. En arrivant dans cet état, un nouvel événement est créé, signalant ainsi le fait qu'une session à demi ouverte a été identifiée. Cet événement peut

```
use netstat ;
scenario halfopentcp(int timeout)
{
   IPAddress victim_addr ;
   Port victim_port ;
   IPAddress attacker_addr ;
   Port attacker_port ;
   timer t0 ;
   initial state s0 {}
   transition SYN (s0 -> s1) nonconsuming
   {
      [IP ip [TCP tcp]] :
      (tcp.tcp_header.flags & TH_SYN) &&!(tcp.tcp_header.flags & TH_ACK)
      {
         victim_addr=ip.header.dst ;
         victim_port=tcp.header.dst ;
         attacker_addr=ip.header.src ;
         attacker_port=tcp.header.src ;
      }
   }
   state s1
   {
      { timer_start(t0, timeout) ; }
   }
   transition ACK (s1 -> s0) unwinding
   {
      [IP ip [TCP tcp]] :
      (ip.header.dst==victim_addr) && (tcp.header.dst==victim_port) &&
      (ip.header.src==attacker_addr) && (tcp.header.src==attacker_port) &&
      !(tcp.header.flags & TH_SYN) && (tcp.header.flags & TH_ACK)
   }
   transition RST (s1 -> s0) unwinding
   {
      [IP ip [TCP tcp]] :
      (ip.header.src==victim_addr) && (tcp.header.src==victim_port) &&
      (ip.header.dst==attacker_addr) && (tcp.header.dst==attacker_port) &&
      (tcp.header.flags & TH_RST)
   }
   transition Timed_out (s1 -> s2) consuming
   {
      timer t0 ;
   }
   state s2
   {
      {
         HALFOPENTCP e ;
         e = new HALFOPENTCP(attacker_addr, attacker_port, victim_addr,
         victim_port, start) ;
         enqueue_event(e, HALFOPENTCP, start) ;
      }
   }
}
```

FIG. 3.1 – Session à demi ouverte dans NetSTAT.

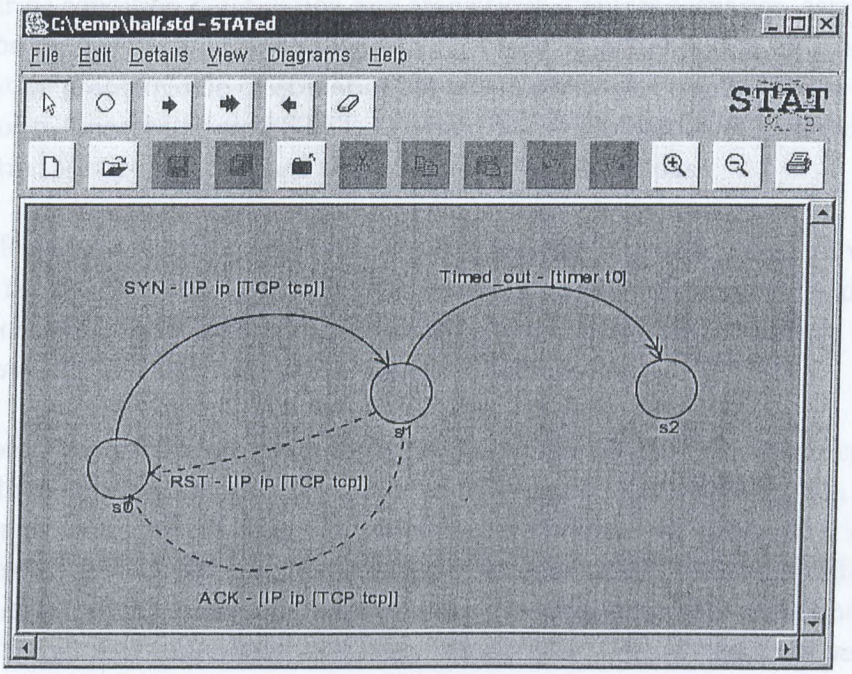

FIG. 3.2 – Session à demi ouverte dans l'outil STATed.

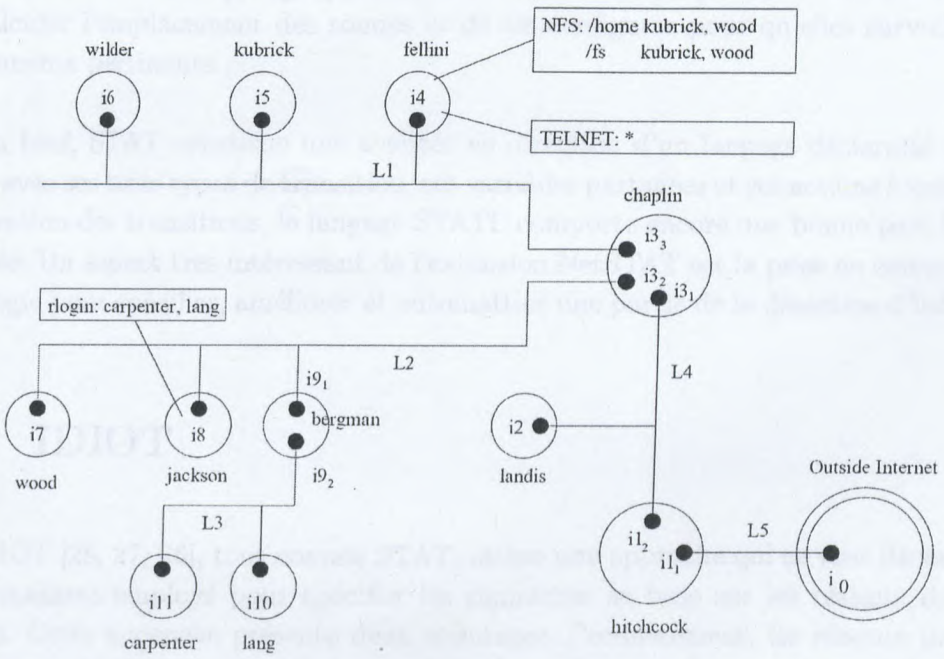

FIG. 3.3 – Modélisation d'un réseau avec NetSTAT.

alors servir de condition d'activation aux transitions d'autres scénarios, servant ainsi à la détection, par exemple, d'une attaque *SYN-Flood*. À la figure 3.2, on peut voir une représentation graphique de ce scénario, obtenue à l'aide de l'outil STATEd, un éditeur de scénarios téléchargeable avec STAT. Les conditions d'activation et les actions à poser peuvent être spécifiées en cliquant sur les états et les transitions.

Les différents systèmes en cours d'exécution peuvent interagir entre eux via des variables partagées. Pour chaque état et pour chaque transition, il est possible de spécifier des assertions qui doivent être vérifiées pour que la transition soit activée de même que des actions à effectuer une fois que la transition a été activée.

Un aspect intéressant de l'extension NetSTAT est qu'elle permet de spécifier la topologie du réseau afin d'améliorer la détection d'intrusion. On peut voir, à la figure 3.3, un exemple de telle modélisation. Un réseau est vu comme un ensemble d'interfaces, alors que les hôtes et les liens de données sont vus comme deux partitions différentes de cet ensemble. La spécification comprend en outre des renseignements sur ces hôtes et ces interfaces tels que leurs adresses matérielles et logicielles. Ces informations permettent, par exemple, de détecter des attaques d'usurpation d'identité en calculant le chemin par lequel deux hôtes sont supposés communiquer. Par exemple, à la figure 3.3, Wilder ne devrait pas voir passer de message en provenance de Carpenter pour Lang. Finalement, la spécification de la topologie permet, en fonction des attaques que l'on veut détecter, de calculer l'emplacement des sondes et de les configurer pour qu'elles surveillent les événements pertinents.

En bref, STAT constitue une avancée en direction d'un langage déclaratif. Cependant, avec ses trois types de transition, ses variables partagées et ses actions à exécuter à l'activation des transitions, le langage STATL comporte encore une bonne part d'impérativité. Un aspect très intéressant de l'extension NetSTAT est la prise en compte de la topologie pour spécifier, améliorer et automatiser une partie de la détection d'intrusion.

3.2 IDIOT

IDIOT [28, 27, 26], tout comme STAT, utilise une approche qui se veut déclarative. Le formalisme employé pour spécifier les signatures se base sur les réseaux de Petri colorés. Cette approche présente deux avantages. Premièrement, les réseaux de Petri permettent de bien représenter le parallélisme et le synchronisme, deux phénomènes bien présents en détection d'intrusion. Avec STAT, le parallélisme est sous-entendu par l'exécution parallèle des systèmes de transition, alors que le synchronisme est représenté

```
01. extern int RLOGIN_PORT_CLIENT, RLOGIN_PORT_SERV,
02.       print_tcp_conn(int, int);

03. pattern TCP_Conn_Mon "Monitor rlogin connections" priority 10
04.    int FROM_PORT, FROM_HOST;
05.    int TO_PORT, TO_HOST;
06.    state start;
07.    nodup state after_syn, after_syn_ack;
08.    state after_ack;
09.    post_action{print_tcp_conn(FROM_HOST, TO_HOST);}

10.    neg invariant first_inv
11.       state inv_start, inv_final;
12.       trans rst(TCP)
13.          <- inv_start;
14.          -> inv_final;
15.          |_{this[RST] = 1 && TO_HOST = this[FROM_HOST] &&
16.          this[TO_HOST] = FROM_HOST;}
17.       end rst;
18.    end first_inv

19.    trans tcp_1(TCP) /* TCP is the event type of the transition */
20.       <- start;
21.       -> after_syn;
22.       |_ {this[SYN] = 1 && this[ACK] = 0 &&
23.       FROM_PORT = this[FROM_PORT] &&
24.       this[TO_PORT] = RLOGIN_PORT_SERV &&
25.       FROM_HOST = this[FROM_HOST] && TO_HOST = this[TO_HOST];}
26.    end tcp_1;

27.    trans tcp_2(TCP)
28.       <- after_syn;
29.       -> after_syn_ack;
30.       |_ { this[SYN] = 1 && this[ACK] = 1 &&
31.       (this[FROM_PORT] = RLOGIN_PORT_SERV)&&
32.       (this[TO_PORT] = FROM_PORT) &&
33.       (this[FROM_HOST] = TO_HOST) && (this[TO_HOST] = FROM_HOST);
34.       }
35.    end tcp_2;

36.    trans tcp_3(TCP)
37.       <- after_syn_ack;
38.       -> after_ack;
39.       |_{ this[SYN] = 0 && this[ACK] = 1 &&
40.       (this[FROM_PORT] = FROM_PORT) &&
41.       (this[TO_PORT] = RLOGIN_PORT_SERV) &&
42.       (this[FROM_HOST] = FROM_HOST) && (this[TO_HOST] = TO_HOST);
43.       }
44.    end tcp_3;
45. end TCP_Conn_Mon;
```

FIG. 3.4 – Session TCP dans IDIOT.

par l'utilisation des variables partagées. Le deuxième avantage de l'approche choisie réside dans le choix de réseaux *colorés*. Les réseaux colorés permettent, comme avec STAT, de définir des patrons d'événements reliés les uns aux autres. Par exemple, ils permettent de spécifier que deux paquets différents doivent avoir les mêmes adresses IP sources et destination.

La documentation sur IDIOT est plutôt rare, et il fait partie des systèmes qu'il ne nous a pas semblé possible de télécharger. Ce que nous avons à notre disposition est un nombre assez restreint d'exemples tirés d'articles. L'exemple de la figure 3.4 est tiré de [26]. Il montre comment on peut utiliser, dans le système IDIOT, les réseaux de Petri colorés pour spécifier l'établissement d'une session TCP. Aux lignes 6,7, et 8 se trouvent les déclarations des états. Comme l'établissement d'une session TCP se fait en trois étapes, on a besoin de quatre états pour la spécifier. Le mot-clé `nodup`, à la ligne 7, signifie que lorsqu'une transition sortante est activée, l'état n'est pas dupliqué. On obtient donc ici un résultat semblable aux transitions *consuming* de STAT, sauf que la spécification se fait au niveau des états plutôt qu'au niveau des transitions. La ligne 9 spécifie quelle est l'action à effectuer lorsque l'état final a été atteint. Les lignes 10 à 18 spécifient un invariant qui doit être vérifié par le système. La notion d'invariant, en IDIOT, s'apparente à la notion de transition *unwinding* en STATL. La spécification d'un invariant prend la forme d'un réseau de Petri, avec ses propres états et ses propres transitions. À chaque fois qu'un jeton sort de l'état initial, une copie en est placée dans l'état initial de l'invariant. Dans l'exemple qui nous occupe, l'invariant nous permet de spécifier qu'aucun paquet RST ne doit être envoyé pendant l'établissement d'une session TCP. Les lignes 19 à 26, 27 à 35, et 36 à 45 spécifient chacune des transitions. Les deux premières lignes indiquent les états entrants et sortants de la transition, alors que le reste spécifie les conditions devant être respectées par le jeton pour que la transition soit activée.

En résumé, le système IDIOT semble présenter certains avantages au niveau de la représentation du parallélisme et du synchronisme. Cependant, sa dépendance envers un autre langage, qui se matérialise par la présence du mot-clé `extern`, en fait un système incomplet qui devrait finalement plutôt être vu comme une librairie complémentaire à un système déjà existant. De plus, la présence du mot-clé `nodup` nous amène au même reproche que nous avons fait à STAT quant à la nature déclarative du langage. Finalement, l'absence dans la littérature d'une documentation complète du langage, ne serait-ce que de sa syntaxe, nous empêche de porter un jugement éclairé sur ses forces et ses faiblesses.

$$p := e(x_1, \ldots, x_n)|cond \mid !p \mid p_1; p_2 \mid p_1||p_2 \mid p^* \mid p \textbf{ within } [t_1, t_2]$$

TAB. 3.1 – Syntaxe des patrons dans BSML.

3.3 BSML

Le langage BSML (*Behavioral Specification Monitoring Language*) [48] est lui aussi à mi-chemin entre le déclaratif et l'impératif. Le formalisme utilisé pour représenter les comportements à identifier est celui des expressions régulières pour les événements (ERE). Les différences entre les expressions régulières conventionnelles et les ERE sont la présence de prédicats du premier ordre pour relier les événements entre eux et la possibilité de spécifier des contraintes de temps-réel.

La syntaxe des ERE utilisées dans le langage BSML se trouve à la table 3.1. Un patrons d'événements est soit un événement primitif, paramétré par $(x_1, \ldots, x_n)$ et respectant la condition *cond*, soit la non-occurrence d'un patron donné ($!p$), soit le patron p_1 suivi immédiatement du patron p_2 ($p_1; p_2$), soit le patron p_1 ou bien le patron p_2 ($p_1||p_2$), soit la répétition un nombre indéterminé de fois du patron p (p^*), soit le patron p devant survenir dans l'intervalle de temps $[t_1, t_2]$ (p **within** $[t_1, t_2]$). L'expression $p..q$ est utilisée pour abréger $p; (!(p||q))^*; q$. Aussi, on utilise les expressions p **over** t et p **within** t pour abréger respectivement p **within** $[t, \infty]$ et p **within** $[0, t]$.

Une spécification BSML ne comporte cependant pas que des ERE. Une spécification BSML est un ensemble de déclarations de variables ainsi que de couples $p \rightarrow act$, où act est une action à poser chaque fois que le patron p est reconnu. Cette action peut être un simple archivage ou encore la manipulation (incrémentation, décrémentation) d'un compteur. Plusieurs compteurs peuvent aussi être regroupés dans une table. La notion de compteur en BSML est assez originale car elle permet de donner valeur plus élevée aux événements plus récents. Les conditions d'activation des actions associées aux compteurs ne sont alors plus nécessairement définies sur le nombre d'événements identifiés, mais sur le poids pondéré de ces événements.

L'exemple de la figure 3.5 est tiré de [48]. Cette signature sert à détecter une inondation TCP SYN. Aux lignes 1 à 3, on définit un prédicat sur deux paquets TCP servant à vérifier si ceux-ci appartiennent à la même session (dans des directions opposées, i.e. avec les adresses IP et les ports inversés). À la ligne 5, on définit l'événement `tcp_syn(p)` comme étant la réception d'un paquet (`rx(p)`) tel que le drapeau `tcp_syn` est activé,

```
01.same_session(p, q) =
02.    p.daddr=q.saddr && p.tcp_dport=q.tcp_sport &&
03.    p.saddr=q.daddr && p.tcp_sport=q.tcp_dport
04.
05.event tcp_syn(p) = rx(p)| p.tcp_syn && !p.tcp_ack
06.event tcp_synack(p, q) = tx(q)| same_session(p,q) && q.tcp_syn && q.tcp_ack
07.event tcp_ack(q, r) = rx(r)| same_session(q,r) && !r.tcp_syn && r.tcp_ack
08.
09.Table neptune(
10.    (unsigned int, unsigned short) /*key : (IP address,port)*/
11.    1000, 120, /*size 1000, window 120 seconds */
12.    4, 1, neptuneBegin, neptuneEnd /* thresholds and associated functions*/
13.)
14.
15.(tcp_syn(p1)..tcp_synack(p1,p2)) ;(( !tcp_ack(p2,p3))* over 60) ->
16.    neptune.inc((p1.daddr, p1.tcp_dport))
```

FIG. 3.5 – Attaque TCP SYN-Flood dans BSML.

mais pas le drapeau `tcp_ack`. À la ligne 6, on définit l'événement `tcp_syn_ack(p,q)` comme étant l'envoi, en direction opposée, d'un paquet ayant les drapeaux `tcp_syn` et `tcp_ack` activés. Finalement, à la ligne 7, on définit l'événement `tcp_ack(q, r)` comme étant la réception, en direction opposée au `syn_ack`, d'un paquet ayant seulement le drapeau `tcp_ack` d'activé. Aux lignes 9 à 13, on déclare la table `neptune` qui servira à compter les occurences de poignées de main non-complétées. La clé de cette table est le couple adresse-port destination du paquet `syn`. La fonction `neptuneBegin` est appelée, par exemple pour lancer une alerte, chaque fois que 4 incrémentations sur une entrée de la table sont faites dans un intervalle de 120 secondes. Cet incrémentation est elle-même effectuée chaque fois que les deux premières étapes d'une poignée de main TCP sont effectuées mais que la troisième ne survient pas dans les 60 secondes suivant la deuxième (lignes 15 et 16).

Le langage BSML est un langage compilé, et les ERE sont transformées en machines à états étendues. La différence entre une machine à état étendue et une machine à états conventionnelle est la même qu'entre des réseaux de Petri conventionnels et des réseaux de Petri colorés : les états de la machine contiennent des variables qui permettent de relier entre eux les symboles lus par la machine. Bien entendu, les symboles de l'alphabet (événements primitifs) sont donc eux aussi caractérisés par des variables.

Contrairement aux langages étudiés jusqu'à maintenant, le langage BSML permet non-seulement de spécifier les comportements qui ne doivent pas survenir, mais aussi les comportements qui doivent survenir, lançant ainsi des alarmes lorsque le comportement observé dévie de ce qui est attendu. Le langage BSML permet ainsi une certaine forme de détection d'anomalies, non-pas basée sur des méthodes statistiques et l'établissement empirique d'un profil normal, mais sur la spécification explicite de ce profil. L'avantage

principal de cette démarche, selon les auteurs, est la détection d'attaques inconnues. Les auteurs jugent aussi que cette démarche diffère suffisamment des démarches statistiques et à base de scénarios pour en faire une troisième catégorie : détection d'intrusions à base de *spécifications*. Dans [49], on trouve une spécification complète de `ftpd` faite avec BSML. BSML peut donc servir non-seulement au niveau réseau, mais aussi au niveau des applications s'exécutant sur un système donné.

Les actions effectuées lors de la reconnaissance de patrons peuvent dépasser le simple archivage : il peut aussi s'agir d'instructions visant à protéger le système qui se fait attaquer. BSML permet ainsi non-seulement une détection d'intrusion en temps réel, mais aussi une prévention d'intrusion.

En résumé, BSML est un langage simple et concis basé sur un formalisme couramment utilisé par les informaticiens : celui des expressions régulières. Ce langage est compilé vers un autre formalisme qui le rend comparable à STATL et à IDIOT : celui des machines à état étendues. Il dispose de primitives permettant d'exprimer des contraintes temps-réel posées sur les événements et de relier ceux-ci entre eux par la valeur de leurs attributs. Parmi les trois langages étudiés dans ce chapitre, il est celui qui se rapproche le plus d'un langage déclaratif car les actions posées lors de la reconnaissance de patrons n'influencent pas la reconnaissance ultérieure d'autres patrons. Cependant, les actions posées par les compteurs indiquent qu'ils jouent eux aussi un rôle important dans l'expressivité du langage et que les expressions régulières à elles seules ne sont pas suffisantes pour exprimer les propriétés voulues. Il s'agit cependant d'un bon compromis entre expressivité et complexité algorithmique.

Chapitre 4

Systèmes experts

Un système expert est un logiciel capable d'accumuler et d'inférer de la connaissance. De plus, les systèmes experts sont souvent appelés à prendre des décisions. Un exemple célèbre de système expert est celui de MYCIN, mis au point dans les années 70 afin d'aider les médecins à poser des diagnostics sur les maladies infectieuses du sang. Le système pose des questions au médecin quand aux symptômes de la maladie, et pose un diagnostique à l'aide d'une base de règles ayant été préalablement été établie par d'autres médecins. L'objectif d'un système expert est de simuler le raisonnement d'un expert dans un domaine particulier de connaissance. Peu importe le domaine, l'hypothèse de base faite sur les raisonnements effectués par les experts est que ceux-ci sont généralement de la forme SI... ALORS.... Par exemple : SI il y a de la fumée ALORS il y a du feu. Cette règle, ajoutée à la règle SI il y a du feu ALORS on doit sonner l'alarme d'incendie, permet au système expert de recommander de sonner l'alarme d'incendie lorsqu'il y a du feu.

Les systèmes experts en détection d'intrusions sont intéressants pour deux raisons : leur capacité d'accumuler et d'inférer de la connaissance permet une détection plus précise (diminution des faux-positifs/négatifs), et leur capacité à prendre des décisions permet de réagir aux attaques identifiées de façon automatisée. Dans ce chapitre, nous donnerons deux exemples de systèmes experts employés en détection d'intrusion : P-BEST et Lambda. P-BEST est un système expert ayant été employé pendant plusieurs années dans le domaine de la recherche en détection d'intrusions, alors que LAMBDA a d'abord été conçu comme un langage ayant pour objectif de permettre aux experts du domaine de transcrire leurs connaissances dans un langage simple.

4.1 P-BEST

P-BEST (Production-Based Expert System Toolset) [30, 2, 1] est en fait un ensemble d'outils permettant de développer un système expert. Il ne s'agit pas d'un système expert au sens propre du terme car, en théorie, un systèmes expert comporte trois composantes principales : un moteur d'inférence, un moteur de décision et une base de connaissance (qui elle même se divise en faits et en règles d'inférences). Généralement, la base de connaissances contient des informations dès le début. Par exemple, dans le cas d'un système expert médical, celle-ci peut contenir les symptômes de la grippe intestinale. La base de connaissances de P-BEST, quant à elle, est complètement vide au moment de l'amorçage. Ce fait ne nous empêche cependant pas de le considérer comme un système expert.

D'un point de vue algorithmique, P-BEST fonctionne en chaînage avant. Ceci signifie que, à chaque ajout d'un nouveau fait, l'ensemble des règles d'inférence est parcouru et que toute l'information déductible de ce fait est ajoutée à la base de connaissance. La base de faits contient donc à la fois les faits de base et les faits calculés. Cette philosophie s'oppose au chaînage arrière, qui consiste à emmagasiner seulement les faits de base, et à inférer sur demande. En détection d'intrusion, où chaque événement signalé correspond à un fait, et où les conclusions critiques doivent être tirées aussitôt que possible, le chaînage avant s'impose donc comme la voie à suivre. De plus, le chaînage avant permet à P-BEST de conserver un minimum d'information en ne conservant dans sa base de faits que ceux qui ont été déduits et en éliminant ceux correspondant aux événements.

P-BEST fournit un langage compilé en C qui offre la possibilité de faire appel à des routines programmées dans ce langage, conservant ainsi une certaine simplicité tout en permettant un maximum de souplesse. Un exemple de règle d'inférence utilisée par P-BEST se trouve à la figure 4.1. Aux ligne 1 à 3, on voit comment le langage fournit par P-BEST permet de déclarer des types d'événements. Les paramètres `#10 ;*` indiquent le niveau de priorité de la règle et le fait que la même règle peut être appliquée plusieurs fois à un même événement. Les niveaux de priorité permettent de spécifier que certaines règles doivent être exécutées avant d'autres. L'opérateur *, indiquant que la règle peut être rappelée plusieurs fois, peut causer des boucles infinies à l'exécution du système s'il n'est pas utilisé correctement. À chaque fois que cet opérateur est utilisé, on doit s'assurer que l'événement sera détruit par au moins une des règles. L'opérateur -, utilisé à la ligne 9, permet justement d'effectuer cette tâche. Il permet de retirer des faits de la base de connaissances; en particulier, il permet aussi de retirer des événements. Chaque événement doit normalement être soigneusement effacé de la base de connaissance afin d'éviter un engorgement de la mémoire utilisée. On doit aussi s'assu-

```
01. ptype[event event_type :int, return_code :int,
02.    username :string, hostname :string]
03. ptype[bad_login username :string, hostname :string]
04.
05. rule[Bad_Login(#10;*) :
06.   [+e :event| event_type == login, return_code == BAD_PASSWORD]
07. ==>
08.   [+bad_login| username = e.username, hostname = e.hostname]
09.   [-|e]
10.   [!|printf("Bad login for user %s from host %s\n",
11.      e.username, e.hostname)]
12. ]
```

FIG. 4.1 – Exemple de règle d'inférence dans P-BEST.

rer que la règle supprimant l'événement aie la priorité la plus basse, afin d'éviter que celui-ci ne soit supprimé avant que toutes les règles en ayant besoin aient pu l'utiliser. Finalement, l'opérateur + doit être vu comme un quantificateur existentiel, l'opérateur ! permet de faire un appel à une routine externe, et l'opérateur ==> sépare la prémisse des conclusions.

D'autres opérateurs sont aussi disponibles dans le langage de P-BEST. Par exemple, il existe des opérateurs permettant d'activer ou de désactiver dynamiquement des règles d'inférence. P-BEST peut donc prendre des décisions à propos de son propre fonctionnement. Le principal intérêt de cette fonctionnalité est de permettre une forme d'optimisation en désactivant des règles jugées temporairement non pertinentes. Il est aussi possible, afin de simplifier l'écriture de certaines règles, de modifier les champs de certains faits. Ceci permet d'émuler la notion de variable, facilitant ainsi, par exemple, la définition de compteurs. Finalement, le langage fournit des opérateurs permettant de marquer ou de démarquer des événements, permettant ainsi à certaines règles de laisser des traces qui peuvent être utilisées par d'autres règles.

En résumé, le système P-BEST permet de développer des systèmes experts dans un langage compilé qui permet une interopérabilité simple et efficace avec d'autres langages, et par conséquent d'autres systèmes. D'un point de vue abstrait, les fonctionnalités de base d'un système expert, soient l'accumulation et l'inférence de connaissance, de même que l'aide à la prise de décision, sont tout à fait souhaitables en détection d'intrusion et P-BEST les implémente très bien. Cependant, le langage de P-BEST n'est peut-être pas des plus appropriés pour la gestion de flots d'événements. En effet, l'association fait-événement respecte bien le paradigme des systèmes experts, mais certains aspects du langage et de l'implémentation dépassent le cadre des systèmes experts et demandent une utilisation consciente des algorithmes utilisés en arrière-plan. Par exemple, le chaînage avant force l'utilisateur à s'assurer de donner les bons ni-

$$e := a \mid \mathbf{0} \mid e_1; e_2 \mid e_1 | e_2 \mid \overline{e} \mid e_1 ? e_2 \mid e_1 \& e_2$$

TAB. 4.1 – Syntaxe du calcul d'événements.

veaux priorité aux différentes règles. De plus, la suppression des événements qu'il n'est plus nécessaire de conserver en mémoire doit être faite explicitement, ce qui complique (inutilement) la rédaction de règles. Finalement, les notions de règles répétables, de marquage d'événements, et d'activation/désactivation de règles donnent à P-BEST des possibilités pratiquement algorithmiques qui dépassent le cadre strict des systèmes experts. P-BEST nous convainc donc que les caractéristiques principales d'un système expert sont souhaitables en détection d'intrusion, mais l'implémentation suggère que le paradigme pur des systèmes experts n'est pas parfaitement adapté à la tâche.

4.2 LAMBDA

LAMBDA est un langage abstrait permettant de décrire des scénarios d'attaque sans se soucier des détails de détection. La description qui en est donnée dans [11] est détachée des détails d'implémentation, et repose sur l'hypothèse que d'autres outils d'audit (dont des systèmes de détection d'intrusion) fournissent un flux d'événements bas niveau à analyser. LAMBDA n'est pas décrit par ses auteurs comme un langage de spécification de système expert, mais comporte tout de même les trois caractéristiques que nous avons identifiées comme étant désirables d'un système expert, soient l'accumulation et l'inférence de connaissance, de même que la possibilité de prendre des décisions et de réagir automatiquement. De plus, comme LAMBDA est spécialement conçu pour la description d'attaques, il comporte des primitives spécialement reliées à ce domaine. Par exemple, il permet de définir un scénario d'attaque de deux points de vue différents : celui de l'attaquant et celui du système de détection. Bien que les deux points de vue soient en général très semblables, il arrive cependant que certaines action effectuées par l'attaquant puissent ne pas être détectables par le système attaqué. Par exemple, il peut s'agir de certains calculs effectués à l'interne.

Le langage LAMBDA offre une solution hybride pour exprimer les scénarios d'attaque, et repose sur trois langages différents, dépendamment du niveau auquel on considère les choses. Pour représenter la connaissance acquise, on utilise la logique du deuxième ordre. On utilise des prédicats du deuxième ordre pour modéliser, par exemple, la connaissance de l'attaquant : le prédicat $knows(A, \phi)$ exprime le fait que

attack attack_name($arg_1, arg_2, \ldots$)
pre : ϕ_{pre}
post : ϕ_{post}
scenario : ϵ_s
where : ψ_s
detection : ϵ_d
where : ψ_d
verification : ϵ_v
where : ψ_v
où ϕ_i est une formule de la logique du deuxième ordre
ψ_i est une formule de la logique du premier ordre
ϵ_i est une formule du calcul des événements

TAB. 4.2 – Syntaxe d'un scénario d'attaque LAMBDA.

l'attaquant A acquiert la connaissance ϕ. Pour représenter les contraintes d'ordonnancement devant être respectées par les différentes étapes d'un scénario d'attaque, on utilise le calcul d'événements [46], dont la syntaxe est donnée à la table 4.1. Un événement peut être soit un événement primitif (a), soit l'événement nul (**0**), soit une séquence d'événements ($e_1; e_2$), soit l'exécution parallèle de deux événements ($e_1|e_2$), soit l'absence d'événements ($\overline{e}$), soit le choix non-déterministe d'événements ($e_1?e_2$), soit l'exécution synchronisée d'événements ($e_1\&e_2$). L'action nulle, combinée au choix non-déterministe, permet de représenter un événement facultatif $[e] \stackrel{\text{def}}{=} e \ ? \ \mathbf{0}$. Il est à noter que l'événement primitif a est défini sur un intervalle de temps $[t_1, t_2]$, et non comme étant un événement ponctuel. Pour les événements ponctuels survenant à l'instant t, l'intervalle est $[t, t]$. Finalement, pour exprimer les contraintes autres que l'ordonnancement devant être respectées par les différents événements du scénario, on utilise simplement la logique du premier ordre.

La syntaxe d'un scénario d'attaque exprimé en LAMBDA se trouve à la table 4.2. Le lien avec la base de connaissances se fait via les clauses **pre** et **post**. La clause **pre** exprime les conditions devant être satisfaites par le système avant l'exécution du scénario afin que celui-ci puisse réussir. La clause **post** exprime l'état du système une fois que l'exécution du scénario a réussi. Les clauses **scenario** et **detection** donnent les détails de l'attaque du point de vue de l'attaquant et de l'attaqué. La clause **verification**, quant à elle, indique comment valider l'effet de l'attaque. Les auteurs parlent aussi d'ajouter une clause **reaction**, permettant d'indiquer comment réagir à l'attaque pour se protéger. Finalement, les clauses **where** donnent les relations entre les diverses étapes des scénarios.

Les notions de post et de pre-conditions permettent, en plus de valider l'occurrence d'attaques, de corréler les différents scénarios entre eux. Deux scénarios sont corrélés si les post-conditions de l'un correspondent aux pre-conditions de l'autre. Cette notion de corrélation permet d'une part de factoriser les scénarios d'attaques lorsqu'un objectif intermédiaire peut être atteint de différentes façons, et d'autre part de découvrir des scénarios d'attaques insoupçonnés.

En résumé, LAMBDA est un langage qui offre les mêmes avantages qu'un paradigme basé sur un système expert : soient l'accumulation et l'inférence d'information, de même qu'un support à la décision. De plus, l'approche hybride utilisant le calcul d'événements permet d'exprimer des scénarios d'attaque selon une syntaxe claire et concise qui semble au moins aussi satisfaisante sinon plus que celles employées avec les systèmes de transition. Des aspects intéressants de LAMBDA sont la séparation du point de vue de l'attaquant et du point de vue de l'attaqué, de même que les notions de vérification et de réaction. Aussi, la description logique des effets et des prémisses d'une attaque permet de faire un calcul de scénario d'attaque afin d'élaborer de nouveaux scénarios.

Chapitre 5

Logiques temporelles

Les trois derniers travaux que nous présentons, LogWeaver, Monid, et Chronicles, sont ceux qui se rapprochent le plus de ceux que nous avons effectués [9, 8, 7, 35, 34, 36] de par le fait que les langages de spécification utilisés ont été développés à partir de logiques temporelles. LogWeaver utilise une logique temporelle future du premier ordre, alors que Monid utilise une logique temporelle passée et future du premier ordre avec points fixes nommée Eagle et que Chronicles se base sur une logique réifiée. Dans les trois cas, les algorithmes utilisés ont été conçus de façon à pouvoir travailler en ligne (*online*). Grossièrement, un algorithme est dit *en ligne* s'il peut traiter séquentiellement sur un fichier d'audit à partir du début sans avoir à mémoriser tout le fichier. Un exemple d'algorithme qui n'est pas en ligne est celui décrit dans [45], où le traitement s'effectue à partir de la fin du fichier. Un tel algorithme est appelé, dans la littérature anglaise, *offline*.

5.1 LogWeaver

L'article sur LogWeaver [44] est divisé en deux parties. Premièrement, les auteurs montrent comment on peut utiliser une logique temporelle classique pour effectuer de la détection d'intrusion sur des fichiers d'audits. Ils montrent ensuite les faiblesses de cette logique et en proposent une autre moins orthodoxe, mais qu'ils considèrent comme plus appropriée. Nous mettrons ici le focus sur la première logique, car elle est plus proche de celles que nous avons déjà vues et elle nous permet de voir comment on peut effectuer de la vérification sur une logique du premier ordre.

$$F ::= A \mid \neg F \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \diamond F$$

TAB. 5.1 – Syntaxe de la première logique de LogWeaver.

Le modèle sur lequel travaille LogWeaver est une séquence finie ou infinie d'enregistrements, appelé indistinctement *log*, ou *trace*. Un enregistrement est défini comme étant une fonction d'un ensemble d'étiquettes vers un ensemble de valeurs, lesquelles sont considérées comme étant des chaînes de caractères.

La syntaxe des formules utilisées dans la première logique abordée est présentée à la table 5.1. À première vue, il s'agit d'un sous-ensemble de LTL. Les opérateurs **U** et ◯ ont été omis car, selon les auteurs, ces derniers ne se révèlent pas d'une grande utilité pour la détection d'intrusion. Il est à noter que l'opérateur $\diamond$ utilisé ici exprime un futur strict. La principale différence entre cette logique et LTL se trouve au niveau du type de la relation de satisfaction et de la sémantique attribuée aux formules atomiques A, aussi appelées *patrons d'événements*. Un exemple de patron d'événements est `{id=X, action="creat", object="/usr/spool/mail/root$"}`. Ce patron filtre tous les événements dont la valeur du champs `id` est égale à celle de la variable X, et dont la valeur des champs `action` et `object` sont respectivement égales à `"creat"` et `"/usr/spool/mail/root$"`.

Étant donné une trace σ et une formule F, l'ensemble des éléments satisfaisant F est alors défini non-pas seulement comme étant un sous-ensemble d'enregistrements de σ, mais comme un ensemble de couples (s, ρ), où s est un enregistrement de σ et ρ est un environnement. Un *environnement* est une fonction attribuant des valeurs aux variables se trouvant dans les patrons d'événements d'une formule. C'est exactement cette notion d'environnement qui justifie l'emploi d'une logique du premier ordre. Ce sont les environnements qui permettent de relier les divers enregistrements entre eux en unifiant les valeurs de certains de leurs champs.

L'algorithme de vérification utilisé pour cette logique est semblable à celui utilisé pour RUSSEL (section 2.1). On travaille cependant avec deux ensembles plutôt que trois, l'ensemble *Cmp* n'ayant aucun sens ici. Pour vérifier une formule F, on initialise donc l'ensemble *Cur* à $\{F\}$ et l'ensemble *Nxt* à $\emptyset$. Ensuite, pour chaque enregistrement, on traite chacune des formules de l'ensemble *Cur*. Ce traitement peut soit amener à ajouter d'autres formules dans l'ensemble *Cur* (dans le cas de formules de la forme $F_1 \wedge F_2$, $F_1 \vee F_2$, ou $\neg F$), soit à ajouter des formules dans l'ensemble *Nxt* (dans le cas des formules de la forme $\diamond F$). Il est à noter que dans le dernier cas, on utilise l'identité

$\diamond F \Leftrightarrow \bigcirc F \vee \bigcirc \diamond F$. En plus de manipuler ces ensembles, il faut aussi ajouter les structures de données nécessaires au chaînage des formules et à la manipulation des environnements. Le lecteur intéressé aux détails de cet algorithme est référé à [44].

Les problèmes reliés à cette logique se divisent en deux catégories. La première catégorie de problèmes concerne le traitement des événements répétés, c'est-à-dire la reconnaissance d'un nombre déterminé d'événements semblables. En plus de ralentir considérablement l'algorithme de vérification en augmentant rapidement et de façon démesurée le nombre de formules à vérifier, le nombre d'alarmes lancées est lui aussi beaucoup trop grand. Ceci est dû au fait que la reconnaissance d'une suite de n événements sur une trace en contenant m lance $\binom{m}{n}$ alarmes, alors qu'en fait une seule pourrait suffire.

La deuxième catégorie de problèmes concerne l'expressivité de la logique, qui est jugée à la fois trop et pas assez expressive. Elle est trop expressive car elle permet la combinaison booléenne de formules de la forme $\diamond F$, qui sont jugées par les auteurs comme étant rarement utiles en pratique. De plus, la présence de négations est jugée comme étant superflue dans la plupart des cas. Ces deux familles de formules, de plus, contribuent chacune de leur façon à la complication de l'algorithme de vérification. Elle n'est pas suffisamment expressive car elle ne permet pas de spécifier naturellement des propriétés de comptage (autrement qu'en écrivant des formules de la forme $F \wedge \diamond(F \wedge \diamond(F \wedge \ldots \wedge \diamond F)))$, ni de parité. Les propriétés de parité sont celles où une action peut annuller temporairement l'effet d'une autre, comme dans le cas de l'ouverture et de la fermeture d'une session TCP.

Pour résoudre ces problèmes, les auteurs suggèrent d'utiliser une autre logique, moins classique, inspirée de ETL [54]. Nous ne nous étendrons pas ici sur les détails de ETL, mais nous nous contenterons de dire que cette dernière a la particularité d'offrir la possibilité d'inclure des automates dans la définition des formules. Une partie de la sémantique des formules est donc définie par la notion d'acceptation d'une chaîne par un automate. Cette solution hybride présente des avantages au niveau de l'expressivité, mais les spécifications obtenues, comme le remarquent les auteurs, commencent à ressembler étrangement à celles de STATL.

En résumé, le travail de M. Roger et J. Goubault-Larrecq est très riche d'un point de vue théorique. L'article [44] comportent plusieurs résultats théoriques qui n'ont pas été cités ici, mais qui sont tout de même très intéressants. De plus, il représente un cas complet d'étude d'utilisabilité d'une logique temporelle classique du premier ordre pour résoudre un problème de vérification en ligne. Les algorithmes utilisés sont présentés en détails et les résultats obtenus sont commentés avec objectivité. Plusieurs problèmes

$$
\begin{array}{lcl}
S & ::= & D\ O \\
D & ::= & R^* \\
O & ::= & M^* \\
R & ::= & \{\mathbf{max}|\mathbf{min}\}\ N(T_1x_1,\ldots,T_nx_n) = F \\
M & ::= & \mathbf{mon}\ N = F \\
T & ::= & \mathbf{Form} \mid \textit{primitive type} \\
F & ::= & \textit{expression} \mid \mathbf{true}|\mathbf{false}| \neg F \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \rightarrow F_2 \\
 & & \mid \bigcirc F \mid \odot F \mid F_1 \cdot F_2 \mid N(F_1,\ldots,F_n) \mid x_i
\end{array}
$$

TAB. 5.2 – Syntaxe des spécifications Eagle.

importants reliés à l'utilisation d'une logique classique ont été relevés, et une solution intéressante a été proposée pour résoudre ces problèmes. Cependant, comme nous l'avons déjà remarqué, cette solution nous fait tendre tranquillement vers un langage à tendance impérative.

5.2 Monid

Le système de vérification en ligne Monid se présente comme étant une alternative à LogWeaver. Ce système utilise une logique temporelle du premier ordre passée et future avec points fixes nommée Eagle [6, 41]. Les auteurs affirment que Eagle est plus expressive que le fragment de LTL utilisé par les auteurs de LogWeaver en ce qu'elle permet d'exprimer des contraintes temps-réel et des propriétés statistiques.

La syntaxe des spécifications Eagle se trouve à la table 5.2. Il s'agit d'une syntaxe plutôt concrète, puisqu'on y spécifie plus que la syntaxe des formules. Une spécification S consiste en une partie de déclarations D et une partie où on définit des observateurs O. D est constitué de zéro ou plus définitions de règles R, et O de zéro ou plus moniteurs M. Les règles et les moniteurs sont nommés N. Le nommage des règles permet la définition de règles récursives. Le domaine lexical T représente les types, qui sont utilisés lors de la définition des règles. Finalement, x_i représente une variable, et *expression* qui s'évalue soit à **vrai** soit à **faux** à chacun des états de la trace.

Les opérateurs $\bigcirc$, $\odot$, désignent respectivement l'événement suivant et l'événement précédent, alors que les opérateurs de points fixes **min** et **max** permettent de définir des formules récursivement. L'opérateur de concaténation $\cdot$ permet de définir des séquences

```
max Always(Form F) = F ∧ ◯Always(F)
min EvTimedLogout(string k, double t, double δ) = (time - t ≤ δ)
        ∧ ((action = LOGOUT ∧ userid = k) ∨ ◯EvTimedLogout(k,t,δ))
mon M = Always(action = LOGIN → EvTimedLogout(userid,time,100))
```

FIG. 5.1 – Exemple de spécification Eagle.

d'événements. Notons aussi que bien que Eagle utilise des variables dans la définition de ses formules, aucune notion d'environnement n'est utilisée pour définir la sémantique. On utilise cependant une notion de substitution textuelle pour définir la sémantique des formules paramétrées $N(F_1, \ldots, F_n)$.

À la figure 5.1, se trouve un exemple de spécification Eagle. Comme d'habitude, on peut définir l'opérateur `Always`, souvent noté $\Box$, à l'aide de l'opérateur de plus grand point fixe et de l'opérateur $\bigcirc$. La définition de la règle `EvTimedLogout` montre comment on peut spécifier qu'un événement doit survenir au plus δ unités de temps après un autre. On procède exactement de la même façon que lorsque l'on définit l'opérateur $\diamond$ à l'aide des opérateurs de plus petit point fixe et $\bigcirc$, sauf que l'on exige en plus que les temps associés aux deux événements ne soient pas séparés de plus de δ unités de temps. Aucun caractère spécial n'est donc donné au temps dans Eagle.

L'algorithme de vérification utilisé par Eagle Flier, le logiciel implantant Eagle, fonctionne à l'aide d'une fonction *eval* qui, étant donné une formule F et un état $\sigma(i)$, donne une nouvelle formule F' telle que $\sigma, i \models F$ si et seulement si $\sigma, i+1 \models F'$. À la fin de la trace, on calcule une fonction booléenne $value(F)$ qui s'évalue à **vrai** si et seulement si le dernier état de σ satisfait F. Donc, pour une trace σ de longueur n et une formule F, on a $\sigma, 1 \models F$ ssi $value(eval(\ldots eval(eval(F, \sigma(1)), \sigma(2)) \ldots, \sigma(n))) =$ **vrai**.

On remarque ici une différence importante entre les hypothèses faites sur le modèle utilisé par LogWeaver et celui de Eagle : les traces vérifiées par Eagle sont de longueur finie, alors que celles sur lesquelles travaille LogWeaver sont définies comme étant finies *ou* infinies. Il est à noter que cette différence reste tout de même très mince car, d'une part, les résultats théoriques sur l'algorithme employé par LogWeaver demandent une trace finie, et d'autre part, les auteurs de [41] citent une méthode, appelée *specifying-bad prefixes* [50], permettant d'exprimer la formule à vérifier de telle sorte que celle-ci devient tôt ou tard vraie ou fausse, peu importe le reste de la trace. L'intérêt de cette technique réside dans le fait qu'elle évite d'avoir à vérifier la satisfiabilité éventuelle d'une formule, problème qui est indécidable dans le cas de Eagle dû à l'utilisation de prédicats du premier ordre.

En résumé, les travaux effectués dans le cadre du développement de Eagle sont plus près de la pratique que ceux effectués pour LogWeaver. Les propriétés intéressantes pour la détection d'intrusion telles que le comptage ou les propriétés de parité s'expriment beaucoup plus naturellement dans Eagle que dans le sous-ensemble de LTL considéré par les auteurs de LogWeaver ou encore la seconde logique proposée, inspirée de ETL. Les algorithmes de vérification utilisés par Eagle sont simples, mais semblent mieux adaptés au cas où la trace d'événements considérée est de longueur finie. Finalement, on trouve dans [41] toute une série d'exemples de scénarios d'attaque allant des plus classiques à d'autres plus originaux pouvant s'exprimer dans Eagle.

5.3 Chronicles

Les deux systèmes que nous venons de présenter ont été construits sur des paradigmes provenant de la théorie de la vérification formelle de modèles. Ils utilisent des logiques qui ont été historiquement développées dans l'intention de vérifier automatiquement certaines propriétés de programmes. Le Model Checking en informatique peut être utile à plusieurs phases du développement, tant à la conception (vérification d'algorithmes), à la programmation (vérification de code), qu'à la phase de tests (validation de logiciels). Le système Eagle, en particulier, a d'abord été développé pour ce type d'usage. L'avantage d'utiliser une logique commune dans un tel cadre d'utilisation est qu'à chaque phase du processus de développement, on peut revérifier les même propriétés avec un minimum d'ajustement. Le Model Checking est très bien adapté aux problèmes booléens : *est-ce que oui ou non telle propriété est satisfaite ?*. Les problèmes du genre : *dans quelle mesure cette propriété est-elle satisfaite ?* ou encore *quelles sont toutes les façons dont cette propriété a été violée ?* présentent peu d'intérêt dans ce domaine et c'est sans doute pourquoi les tentatives d'y appliquer les logiques y ayant été développées peuvent parfois sembler artificielles et tordues.

En intelligence artificielle, les problèmes auxquels on s'attaque sont traditionnellement différents. La nature des problèmes abordés est beaucoup plus vaste, et souvent plutôt éloignée de ceux généralement rencontrés par un informaticien. Par exemple, on pourra être intéressé à analyser les battements cardiaques d'un être humain. Ici, on a un exemple intéressant où tout ce qui compte est la fréquence des événements, ceux-ci étant indistinctibles les uns des autres. On sera intéressé au nombre de battements dans une minute, à l'accélération du rythme, aux moments où de légères anomalies semblent survenir, etc. Dans un tel contexte, le besoin de logiques où le comptage et les intervalles de temps sont élevés au rang de primitives du langage se fait bien sentir, et c'est dans un tel contexte que les logiques réifiées ont été développées. Les logiques réifiées sont des

$hold(P : v; (t_1; t_2))$
$event(P : (v_1; v_2); t)$
$event(P; t)$
$noevent(P; (t_1; t_2))$
$occurs((n_1; n_2); P; (t_1; t_2))$

TAB. 5.3 – Prédicats de réification de Chronicles.

logiques du deuxième ordre où les prédicats du deuxième ordre servent essentiellement à exprimer les instants ou les intervalles de temps où les faits exprimés par les prédicats du premier ordre sont vrais.

Chronicles, introduit dans [15], est un langage basé sur les logiques réifiées permettant de reconnaître des *chroniques* dans un flot d'événements. Il vient avec CRS (Chronicles Recognition System) un vérificateur qui peut fonctionner en ligne. Informellement, une chronique est un ensemble d'événements reliés entre eux par des contraintes temporelles. Dans la littérature, les chroniques sont reliées au calcul des événements présenté plus haut.

La représentation du temps dans Chronicles est discrète et il est donc vu comme une suite ordonnée d'instants dont la résolution est assez fine pour les besoins de l'analyse. Deux événements différents peuvent survenir au même instant, mais lorsque deux ou plusieurs événements identiques surviennent au même instant (selon la résolution choisie), ceux-ci sont fusionnés en un seul événement. En détection d'intrusion, où plusieurs événements identiques peuvent survenir en peu de temps, le choix de la résolution doit donc être fait avec précautions.

L'environnement est décrit par les attributs du domaine. Un *attribut du domaine* est un tuple $P(a_1, \ldots, a_n) : v$, où P est le nom de l'attribut, $a_1, \ldots, a_n$ ses arguments, et v sa valeur. Certains attributs peuvent ne pas avoir de valeur. Ces derniers sont appelés *messages*. Finalement, les *événements* correspondent à des changements de valeurs des attributs du domaine.

Les prédicats de réification de Chronicle sont présentés à la table 5.3. Le prédicat $hold(P : v; (t_1; t_2))$ signifie que l'attribut P conserve la valeur v sur l'intervalle $[t_1, t_2[$. Le prédicat $event(P : (v_1; v_2); t)$ signifie que l'attribut P a passé de la valeur v_1 à v_2 à l'instant t. Le prédicat $event(P; t)$ signifie que le message P est survenu à l'instant t. Le prédicat $noevent(P; (t_1; t_2))$ signifie qu'aucun changement de valeur de l'attribut P n'est survenu durant l'intervalle $[t_1, t_2[$. Finalement, le prédicat $occurs((n_1; n_2); P; (t_1; t_2))$ permet de compter. Il signifie que sur l'intervalle $[t_1, t2[$, l'événement P est survenu de

```
1. chronicle exemple1 {
2.    event(e1,t1) ;
3.    event(e2,t2) ;
4.    event(e3,t3) ;
5.
6.    t1<t2<t3
7.    t3-t2 <= 4
8. }
```

FIG. 5.2 – Exemple de Chronique.

n_1 à n_2 fois.

Un *modèle de chronique* est constitué de cinq choses : i) un ensemble d'instants, ii) un ensemble de contraintes temporelles sur ces instants, iii) un ensemble de patrons d'événements qui représentent les changements de l'environnement auxquels on s'intéresse iv) un ensemble d'assertions qui représentent le contexte dans lequel les événements surviennent, et v) un ensemble d'actions externes à prendre lorsqu'une instance de chronique est reconnue.

Un exemple de modèle de chronique est présenté à la figure 5.2. Il sera reconnu à chaque fois que les messages e_1, e_2 et e_3 surviendront dans cet ordre et que, de plus, le message e_3 ne doit pas survenir plus de 4 unités de temps après e_2.

L'algorithme de reconnaissance des chroniques fonctionne de façon semblable à d'autres que nous avons déjà vus. Au début, un modèle vide est créé. À chaque fois qu'un événement satisfaisant les contraintes exprimées par un modèle de chronique survient, celui-ci est ajouté à l'instance de chronique en cours après duplication de cette dernière. La duplication sert à s'assurer de reconnaître toutes les instances possible. Certaines instances en cours de reconnaissance peuvent être éliminées lorsque des assertions sont violées ou que les contraintes temporelles ne peuvent plus être satisfaites. Il est important, lors de la spécification de chroniques, de s'assurer de spécifier de telles assertions ou contraintes temporelles. Autrement, et particulièrement dans un mode de fonctionnement en ligne, certaines instances peuvent rester indéfiniment en mémoire et même, éventuellement, dépasser les capacités de celle-ci.

Un aspect important de Chronicles est que l'action à prendre lors de la reconnaissance d'une chronique n'est pas nécessairement une simple action d'archivage externe. Il peut aussi s'agir de l'émission d'un nouvel événement, qui peut alors être utilisé soit par une autre chronique en cours de reconnaissance, soit par d'autres instances de la même chronique. Cette fonctionnalité donne à Chronicles une certaine forme d'acquisition de connaissance, celle-ci pouvant être représentée par certains attributs du domaine dont

les valeurs sont tenues à jour par le mécanisme même de reconnaissance de chroniques.

Chronicles n'a pas été conçu spécialement en vue de faire de la détection d'intrusion. À ses débuts, il a été appliqué à l'analyse de journaux d'alarmes d'équipements de télécommunication. Il a par la suite été appliqué à d'autres domaines tels que l'analyse de circulation routière et on lui a même trouvé quelques applications en médecine. Dans [38], on a finalement suggéré une façon de l'utiliser en détection d'intrusions. Bien qu'il soit possible d'utiliser Chronicles pour représenter des scénarios d'attaque, l'utilisation proposée par les auteurs tend vers d'autres objectifs. Principalement, ils proposent d'utiliser Chronicles comme outil d'analyse des journaux d'alarmes de systèmes de détection d'intrusion afin de i) permettre une détection plus précise, ii) réduire le nombre d'alarmes et iii) améliorer la sémantique des alarmes.

L'utilisation de Chronicles peut permettre une détection plus précise. Plus précisément, elle peut aider à diminuer le nombre de faux-positifs en invalidant certaines alarmes. L'invalidation d'alarmes avec Chronicles peut se faire en spécifiant certains contextes dans lesquels certains comportement ne doivent pas être considérés comme dangereux. Par exemple, l'initiation d'une video-conférence entre plusieurs utilisateurs peut sous certains aspects ressembler à un balayage de ports, mais ne doit tout de même pas déclencher d'alarme. L'utilisation de Chronicles peut réduire le nombre d'alarmes, principalement en utilisant le prédicat *occurs*. Certaines attaques, telles que la propagation de vers informatiques, se caractérisent par la répétition à outrance de comportements identiques, et peuvent déclencher un nombre abrutissant et inutilement élevé d'alarmes. Chronicles peut fusionner ces différents événements en un seul. Chronicles peut améliorer la sémantique des alarmes en combinant les différents symptômes d'une même attaque. Mieux encore, certaines attaques ne doivent être considérées comme réussies que si un ensemble bien déterminé d'actions ont été effectuées. Certains systèmes de détection d'intrusion lanceront des alarmes à chacune de ces actions, donnant ainsi l'impression que de nombreuses attaques ont eu lieu alors qu'il s'agit en fait d'une seule. L'utilisation de Chronicles permet de ne déclencher qu'une seule alarme rendant ainsi mieux compte du fait qu'une seule attaque a eu lieu.

Les auteurs de [38], en plus de donner plusieurs exemples où Chronicles peut s'avérer utile dans l'analyse de journaux d'alarmes, montrent comment ce langage peut être utilisé de paire avec M2D2 [39] pour corréler les alarmes avec le contexte. M2D2 est un modèle formel incluant celui de NetSTAT [53]. Il permet de représenter non-seulement la topologie physique du réseau, mais aussi toute sa configuration, les logiciels installés sur chacune des machines, les vulnérabilités de ceux-ci de même que leurs effets, les logiciels de sécurité installés sur chacune des composantes, les attaques détectables par chacun des systèmes de détection d'intrusion, et bien encore. La façon dont les alarmes

sont représentées dans M2D2 sont de plus compatibles avec IDWG [12], un format d'échange d'alarmes standardisé mis au point à l'IETF. L'utilisation d'un tel modèle permet, en plus de corréler différentes alarmes entre elles, de tenir compte dans l'analyse des caractéristiques du système attaqué et même du système de détection d'intrusions ayant signalé l'attaque.

En résumé, Chronicles est un langage de surveillance basé sur une logique réifiée dont la sémantique semble mieux adaptée à la détection d'intrusion que les logiques traditionnellement utilisées en Model Checking. L'ordonnancement des différents événemets est exprimé par un ensemble de contraintes sur les instants leur étant associés plutôt que par des opérateurs dédiés. Les opérateurs spécifiques à Chronicles mettent plutôt le focus sur les différents changements pouvant survenir pendant le processus de reconnaissance d'une chronique. Un de ces opérateurs permet même de compter de façon très naturelle le nombre d'événements semblables. Chronicles a de plus été utilisé dans de nombreux domaines d'application à première vue assez différents les uns des autres. Un des dangers reliés à l'utilisation de Chronicles est l'accroissement exponentiel de l'utilisation de la mémoire lorsque les chroniques ne sont pas spécifiées consciencieusement. Finalement, des travaux ayant été effectués tendent à démontrer que Chronicles peut s'avérer utile en corrélation d'alarmes. La notion de corrélation d'alarmes employée par les auteurs de [38] est cependant légèrement différente de celle utilisée par les auteurs de [11]. Les premiers utilisent le mot corrélation dans un sens plutôt général, désignant par exemples différentes alarmes symptomatiques d'un même phénomène, alors que les deuxièmes parlent plutôt d'une corrélation logique, voire même causale. Le vérificateur de chroniques CRS permet de générer des événements en cours d'exécution. Cette fonctionnalité, jointe à l'utilisation du modèle M2D2, permet de bien mettre les attaques signalées en contexte.

Chapitre 6

Autres approches

La revue que nous avons faite mettait l'accent sur les langages et les paradigmes utilisés pour exprimer les signatures dans un contexte de détection. D'autres langages, comme ADeLe [37], NASL [13] et CASL [47] mettent plutôt l'accent sur le côté attaquant. ADeLe a été développé en parrallèle avec LAMBDA dans le but développer une base de données d'attaques. Une des préocupations des auteurs était de concevoir un langage assez général pour pouvoir représenter, comme LAMBDA, les attaques à la fois du point de vue de l'attaquant et de l'attaqué, de même que de permettre une corrélation logique entre les différentes attaques et de spécifier comment réagir à ces attaques. NASL est un langage de scripts développé pour l'identificateur de failles de sécurité Nessus. Il fournit des primitives permettant d'envoyer des paquets forgés *à la main* sur le réseau et de recevoir d'autres paquets en vue d'effectuer des tests bien précis. CASL a été développé avec des objectifs similaires et offre des fonctionnalités semblables, mais a été conçu avec l'objectif de réaliser des attaques plutôt que de simplement tester les failles.

Une approche dont nous n'avons pas parlé est celle de [43], basée sur un paradigme de grammaires. Les auteurs de cet article proposent un formalisme pour représenter les attaques comprenant entre autres des notions de filtre, de temps-réel, et de séquencement. Ce formalisme vient avec un algorithme de vérification à base de tableaux qui est prouvé complet et cohérent. On trouve aussi dans cet article une discussion intéressante au sujet des problèmes pratiques découlant de la complétude des algorithmes de vérification de signatures dans le cas des langages déclaratifs. Principalement, on soulève le fait que la complétude, bien qu'intéressante d'un point de vue théorique, engendre souvent des problèmes de complexité algorithmique et de surcharge d'alarmes. La méthode proposée par les auteurs permet de *régler* l'algorithme de vérification en définissant une relation d'équivalence sur les alarmes de façon à réduire le nombre d'alarmes tout en

conservant celles qui sont considérées comme étant les plus significatives. Le système ARMD [29], mis au point pour le langage MuSigs, basé sur l'algèbre relationnelle, offre lui aussi la possibilité de régler l'algorithme de vérification selon les besoins particuliers de l'utilisateur.

Plusieurs approches issues de techniques d'intelligence artificielle ont aussi été proposées pour attaquer le problème de la détection d'intrusions. Dans [33], on propose une méthode utilisant les algorithmes génétiques pour apprendre, à partir d'un ensemble d'entraînement, à reconnaître les patrons d'attaque. Cette méthode, bien que comportant une phase d'apprentissage, présente cependant deux différences importantes avec les méthodes basées sur une détection d'anomalies (méthodes statistiques). Premièrement, l'ensemble d'entraînement fourni à l'algorithme doit être étiqueté. Alors que les méthodes basées sur une détection d'anomalies ne prennent en entrée que des fichiers d'audit, la méthode proposée a besoin, de plus, de savoir où se trouvent les attaques dans ce fichier. Deuxièmement, le résultat de la phase d'apprentissage ne donne pas un profil qui doit être considéré comme normal, mais un ensemble de signatures qui peut par la suite être utilisé par un algorithme de reconnaissance de scénarios. Dans [10] et [51], on propose d'autres méthodes basées sur la programmation génétique. La programmation génétique peut être vue comme un cas particulier des algorithmes génétiques où les individus participant aux croisements sont des programmes, et les gênes sont des bouts de code. Encore une fois, ces approches dépendent de la disponibilité de fichiers d'audits étiquetés pour la phase d'apprentissage. Cette contrainte n'est pas des moindres car, depuis les fichiers d'évaluation créés par le laboratoire Lincoln du MIT en 1998 et 1999 [31, 32], peu d'autres fichiers d'audit ou de traces de trafic ont été fournis à la communauté pour permettre de tels travaux.

Conclusion

Dans ce rapport, nous avons étudié 13 systèmes de détection d'intrusion différents, dont 12 sont à base de scénarios. Les langages utilisés pour exprimer les scénarios reposent sur 5 catégories de paradigmes : les langages spécifiques au domaine, les langages impératifs, les systèmes de transition, les systèmes experts, et les logiques temporelles. Nous avons identifié, pour chacun de ces langages, des avantages et des inconvénients qui leur sont propres. Nous donnons à la table 6.1 la liste des dix caractéristiques, parmis celles que nous avons relevé, que nous jugeons les plus importantes pour un système de détection d'intrusions à base de scénarios. Pour chacun des IDS étudiés, nous indiquons les caractéristiques qu'il comporte à la table 6.2.

Comme nous l'avons dit au début du chapitre 5, cette revue de littérature s'insère dans le cadre de travaux que nous effectuons présentement en détection d'intrusions. Dans [9, 8, 7], nous explorons la possibilité d'utiliser une logique temporelle pour représenter à la fois les scénarios d'attaque à plusieurs événements et l'acquisition passive de connaissance sur le réseau. Dans [35, 34], nous proposons un formalisme basé sur les méthodes d'analyse orientée-objet afin de représenter de façon uniforme les paquets circulant sur le réseau, le réseau lui-même, de mêmes que les diverses activités y ayant cours. Finalement, dans [36], nous présentons une méthodologie que nous avons mise au point dans le but de développer une suite de tests pour les systèmes de détection d'intrusions. Ces tests ont pour but de mesurer le taux de faux-positifs/négatifs engendrés par les systèmes de détection d'intrusions.

1) **scénarios à plusieurs événements** Les différents exemples que nous avons étudié nous convainquent que le fait de pouvoir exprimer des scénarios comportant plusieurs événements est nécessaire pour effectuer une détection d'intrusion précise et complète.

2) **non-occurence d'événements** Il arrive parfois que le fait de ne pas observer un événement soit révélateur d'information.

3) **contraintes temps-réel** On doit être capable de spécifier un délai entre les différents événements d'un scénario.

4) **comptage** Dans certains cas, c'est la répétition d'un événement ou d'un scénario donné qui est porteur d'information.

5) **acquisition de connaissance** Le fait de pouvoir extraire et conserver de façon dynamique de l'information des événements observés aide à réduire les faux positifs.

6) **propriétés de parité** Certaines conditions ne sont valides qu'entre deux événements donnés.

7) **gestion des attributs absents ou multiples** Le langage doit prendre automatiquement en compte le fait que certains attributs peuvent être optionnels pour certains événements. Aussi, il est possible qu'un attribut ait une liste de valeurs.

8) **approche déclarative** Un paradigme déclaratif permet, en plus de faciliter la maintenance de la base de signatures, d'effectuer un calcul de corrélation entre les différents scénarios. Ce calcul n'est cependant possible que si le langage est purement déclaratif, au sens de la définition 3.1.

9) **fonctionnement en ligne** L'algorithme de vérification doit être capable de traiter les événements dans l'ordre où ils surviennent, et doit automatiquement les effacer de la mémoire dès qu'ils sont traités.

10) **utilisation bornée de la mémoire** L'algorithme de vérification doit être capable de fonctionner avec une quantité fixe de mémoire qui n'augmente pas avec le nombre d'événements.

TAB. 6.1 – Dix propriétés souhaitables d'un IDS.

nom	1	2	3	4	5	6	7	8	9	10
Snort	N	N	N	N	O	O	N	O	O	O
NeVO	O	N	O	N	O	N	N	O	O	O
ASAX	O	N	O	O	O	O	O	N	N	N
Bro	O	N	O	O	O	O	N	N	O	O
STAT	O	O	O	O	O	O	N	N	O	N
IDIOT	O	N	O	O	O	O	N	N	O	N
BSML	O	O	O	O	N	N	N	N	O	O
P-BEST	O	N	N	N	O	O	N	N	O	N
LAMBDA	O	O	O	N	O	O	N	O	-	-
LogWeaver	O	O	O	N	N	N	N	O	O	N
Monid	O	O	O	O	N	N	N	O	O	N
Chronicles	O	O	O	O	N	N	N	O	O	N

TAB. 6.2 – Tableau récapitulatif des IDS.

Bibliographie

[1] Anderson, Frivold, and Valdes. NIDES : A summary. may 1995.

[2] Anderson, Lunt, Javits, Tamaru, and Valdes. NIDES : software users manual — beta-update release. dec 1994.

[3] Harry Anderson. Introduction to nessus. `http://www.securityfocus.com/infocus/1741`, october 2003.

[4] Harry Anderson. Nessus, part 2 : Scanning. `http://www.securityfocus.com/infocus/1753`, december 2003.

[5] Harry Anderson. Nessus, part 3 : Analysing reports. `http://www.securityfocus.com/infocus/1759`, february 2004.

[6] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification, 2004.

[7] Mathieu Couture. Détection d'intrusions et analyse passive de réseaux. Master's thesis, Université Laval, Département d'informatique et de génie logiciel, 2005.

[8] Mathieu Couture, Béchir Ktari, Frédéric Massicotte, and Mohamed Mejri. A declarative approach to stateful intrusion detection and network monitoring. In *Proceeding of the Second Annual Conference on Privacy, Security and Trust*, Fredericton, New Brunswick, Canada, octobre 2004.

[9] Mathieu Couture, Béchir Ktari, Frédéric Massicotte, and Mohamed Mejri. Détection d'intrusions et acquisition passive d'information. In *Actes de la 3ème Conférence sur la Sécurité et Architectures Réseaux*, La Londe, Côte d'Azur, France, juin 2004.

[10] Mark Crosbie and Eugene H. Spafford. Applying genetic programming to intrusion detection. In E. V. Siegel and J. R. Koza, editors, *Working Notes for the AAAI Symposium on Genetic Programming*, pages 1–8, MIT, Cambridge, MA, USA, 10–12 1995. AAAI.

[11] Frédéric Cuppens and Rodolphe Ortalo. Lambda : A language to model a database for detection of attacks. In *Proceedings of the Third International Workshop on Recent Advances in Intrusion Detection*, pages 197–216. Springer-Verlag, 2000.

[12] H. Debar, D. Curry, and B. Feinstein. Intrusion detection exchange format. `http://www.ietf.org/internet-drafts/draft-ietf-idwg-idmef-xml-12.txt`, july 2004. Internet-Draft.

[13] Renaud Deraison. The nessus attack scripting language reference guide. `http://www.nessus.org/doc/nasl.html`, april 2000.

[14] Renaud Deraison, Ron Gula, and Todd Hayton. Passive vulnerability scanning- an introduction to nevo. `http://www.tenablesecurity.com/papers.html`, august 2003.

[15] Christophe Dousson. *Suivi d'évolutions et reconnaissance de chroniques.* PhD thesis, Université Paul Sabatier de Toulouse, September 1994.

[16] S. Eckmann, G. Vigna, and R. Kemmerer. Statl : An attack language for state-based intrusion detection, 2000.

[17] Fyodor. The art of port scanning. `http://www.insecure.org/nmap/nmap_doc.html`, 1997.

[18] Coretez Giovanni. Fun with packets : Designing a stick. `http://packetstormsecurity.nl/distributed/stick.htm`.

[19] Chris Green and Martin Roesch. Snort users manual 2.1.0 - the snort project, December 2003.

[20] Ron Gula. Correlating ids alerts with vulnerability information. `http://www.tenablesecurity.com/papers.html`, december 2002.

[21] Naji Habra, Baudouin Le Charlier, Abdelaziz Mounji, and Isabelle Mathieu. ASAX : Software architecture and rule- based language for universal audit trail analysis. In *European Symposium on Research in Computer Security (ESORICS)*, pages 435–450, 1992.

[22] Naji Habra, Baudouin Le Charlier, Abdelaziz Mounji, and Isabelle Mathieu. Preliminary report on advanced security audit trail analysis on unix (asax also called satx). Technical report, Institut D'Informatique, FUNDP, rue Grangagnage 21, 5000 Namur, Belgium, September 1994.

[23] Koral Ilgun. USTAT : A real-time intrusion detection system for UNIX. In *Proceedings of the 1993 IEEE Symposium on Research in Security and Privacy*, pages 16–28, Oakland, CA, 1993.

[24] Van Jacobson, Craig Leres, and Steven McCanne. tcpdump man page. `http://www.tcpdump.org/tcpdump_man.html`.

[25] Van Jacobson, Craig Leres, and Steven McCanne. libpcap man page. `http://www.tcpdump.org/pcap3_man.html`, November 2003.

[26] S. Kumar and E. Spafford. A software architecture to support misuse intrusion detection. In *Proceedings of the 18th National Information Security Conference*, pages 194–204, 1995.

[27] Sandeep Kumar. *Classification and Detection of Computer Intrusions.* PhD thesis, Purdue, IN, 1995.

[28] Sandeep Kumar and Eugene Spafford. An Application of Pattern Matching in Intrusion Detection. Technical Report 94-013, Department of Computer Sciences, 1994.

[29] Lin, Wang, and Jajodia. Abstraction-based misuse detection : High-level specifications and adaptable strategies. In *PCSFW : Proceedings of The 11th Computer Security Foundations Workshop.* IEEE Computer Society Press, 1998.

[30] Ulf Lindqvist and Phillip A Porras. Detecting computer and network misuse through the production-based expert system toolset (p-best). In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 146–161, Oakland, California, may 1999. IEEE Computer Society Press, Los Alamitos, California.

[31] Richard Lippmann, David Fried, Isaac Graf, Joshua Haines, Kristopher Kendall, David McClung, Dan Weber, Seth Webster, Dan Wyschogrod, Robert Cunningham, and Marc Zissman. Evaluating intrusion detection systems : The 1998 DARPA off-line intrusion detection evaluation. In *Proceedings of the DARPA Information Survivability Conference and Exposition*, Los Alamitos, CA, 2000. IEEE Computer Society Press.

[32] Richard Lippmann, Joshua W. Haines, David J. Fried, Jonathan Korba, and Kumar Das. The 1999 darpa off-line intrusion detection evaluation. *Comput. Networks*, 34(4) :579–595, 2000.

[33] Ludovic Mé. Gassata, a genetic algorithm as an alternative tool for security audit trails analysis. *First International Workshop on Recent Advances in Intrusion Detection (RAID '98)*, 1998.

[34] Frédéric Massicotte. Using object-oriented modeling for specifying and designing a network-context sensitive intrusion detection system. Master's thesis, Department of Systems and Computer Engineering, Carleton University, September 2005.

[35] Frédéric Massicotte, Mathieu Couture, Lionel Briand, and Yvan Labiche. Context-based intrusion detection using snort, nessus and bugtraq databases. In *Third Annual Conference on Privacy, Security and Trust (PST'2005)*, St. Andrews , New Brunswick, Canada, October 2005.

[36] Frédéric Massicotte, Mathieu Couture, Lionel Briand, and Yvan Labiche. A proof of concept for a context-based intrusion detection system. In *Annual Computer Security Applications Conference (ACSAC'05)*, Tucson , Arizona, December 2005.

[37] 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.

[38] B. Morin and H. Debar. Correlation of intrusion symptoms : an application of chronicles. In *6th International Conference on Recent Advances in Intrusion Detection (RAID 2003)*, volume 2820 of *LNCS*, Pittsburg, September 2003. Springer.

[39] Benjamin Morin, Ludovic Mé, Hervé Debar, and Mireille Ducassé. M2d2 : A formal data model for ids alert correlation. In *5th International Conference on Recent Advances in Intrusion Detection (RAID 2002)*, volume 2516 of *LNCS*, pages 177–198, Zurich, October 2002. Springer.

[40] Abdelaziz Mounji. *Languages and Tools for Rule-Based Distributed Intrusion Detection.* PhD thesis, Facultés Universitaires Notre-Dame de la Paix, Namur, Belgique, 1997.

[41] Prasad Naldurg, Koushik Sen, and Prasanna Thati. A temporal logic based framework for intrusion detection.

[42] Vern Paxson. Bro : a system for detecting network intruders in real-time. *Computer Networks (Amsterdam, Netherlands : 1999)*, 31(23–24) :2435–2463, 1999.

[43] Jean-Philippe Pouzol and Mireille Ducassé. Formal specification of intrusion signatures and detection rules. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW'02)*, page 64. IEEE Computer Society, 2002.

[44] Muriel Roger and Jean Goubault-Larrecq. Log auditing through model checking. In *Proc. 14th IEEE Computer Security Foundations Workshop (CSFW'01), Cape Breton, Nova Scotia, Canada, June 2001*, pages 220–236. IEEE Comp. Soc. Press, 2001.

[45] Grigore Rosu and Klauss Havelund. Synthesizing dynamic programming algorithms from linear temporal logic formulae, 2000.

[46] Fariba Sadri and Robert A. Kowalski. Variants of the event calculus. In *International Conference on Logic Programming*, pages 67–81, 1995.

[47] Inc. Secure Networks. Custom attack simulation language (casl). `http://www.sockpuppet.org/tqbf/casl.html`, 1997.

[48] R. Sekar, Y. Guang, S. Verma, and T. Shanbhag. A high-performance network intrusion detection system. In *ACM Symposium on Computer and Communication Security*, 1999.

[49] R. Sekar and P. Uppuluri. Synthesizing fast intrusion detection/prevention systems from high-level specifications. In *USENIX Security Symposium*, 1999.

[50] K. Sen, G. Rosu, and G. Agha. Online efficient predictive safety analysis of multithreaded programs, 2004.

[51] Dong Song, Malcolm I. Heywood, and A. Nur Zincir-Heywood. A linear genetic programming approach to intrusion detection. In *Genetic and Evolutionary Computation - GECCO-2003*, volume 2724 of *LNCS*, pages 2325–2336, Chicago, 12-16 July 2003. Springer-Verlag.

[52] Diomidis Spinellis and Dimitris Gritzalis. Panoptis : Intrusion detection using a domain-specific language. *Journal of Computer Security*, 10 :159–176, 2002.

[53] Giovanni Vigna and Richard A. Kemmerer. Netstat : A network-based intrusion detection approach. In *ACSAC*, pages 25–, 1998.

[54] Pierre Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1–2) :72–99, 1983.