UNIVERSITÉ DU QUÉBEC À MONTRÉAL

ÉTUDE ET MISE EN ŒUVRE D'UNE MÉTHODE DE DÉTECTION D'INTRUSIONS DANS LES RÉSEAUX SANS-FIL 802.11 BASÉE SUR LA VÉRIFICATION FORMELLE DE MODÈLES

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

ROMDHANE BEN YOUNES

DÉCEMBRE 2007

À Lina et à ma famille.

REMERCIEMENTS

Mes meilleurs remerciements vont naturellement à ceux qui ont contribué le plus directement à la réalisation de ce travail que ce soit par leurs idées ou par leur soutien financier, je nomme : M. Guy Tremblay, M. Guy Bégin, M. Julien Olivain, la Mission Universitaire de Tunisie en Amérique du Nord pour la bourse d'exemption des frais majorés et la Fondation de l'UQÀM pour les différentes bourses que j'ai obtenues durant mes études.

Mes remerciements vont également à tous mes enseignants tout au long de ma scolarité qui ont certainement contribué indirectement par leurs enseignements, je nomme : M. Roger Villemaire, M. Abdellatif Obaïd, M. Imed Gargouri et M. Robert Godin.

Une reconnaissance particulière à M. Kamel Ouachani et à sa femme qui étaient les seules personnes que je connaissais à mon arrivée au Canada. Ils m'ont accueilli, aidé et supporté tout au long de mon séjour. Ils ont été pour moi l'odeur de ma famille. Bref, je leur dois beaucoup. Grâce à eux, j'ai rencontré plusieurs autres personnes adorables. Mes remerciements s'adressent particulièrement à M. Abderraouf et à toute sa famille qui m'ont considéré comme un proche.

Au cours de mon séjour à Montréal, j'ai eu le plaisir de côtoyer et de faire la connaissance de personnes très sympathiques que je voudrais remercier ici rien que pour l'ambiance favorable qu'ils ont instauré et pour la qualité des échanges scientifiques et culturels que nous avons entrepris. Merci à Ajib Wessam, Arsenault Lise, Béjaoui Rym, Belyamani Mohamed, Bisson Sylvie, Boukhris Rihab, Boukhris Taqwa, Chami Aïda, Delicieux Donald, Delicieux Josette, Djanou Claudel, Djoufak Jean-Françoi, El-Boussaidi Ghizlene, Faghihi Usef, Figaro Luc, Gaha Mohamed, Gingras Éric, Hegnievitzki Christophe, Jendoubi Wassim, Jubinville Émanuelle, Kamari Anis, Lazaar Jamal, Le Helley Kévin, Lord Mélanie, Maffezzini Ivan Patrizio, Mili Hafedh, Mojbari Belhssen, Nama Antoine, Naoum Hanae, Nsiempba Jude Jacob, Ouellette Lise, Pagé Christian, Paradis Guy, Salah Aziz, Séguin Normand, Sellaouti Rabiaa Mojbari, Sellaouti Zied, Selsouli Mehdi, Trépanier France, et Vézina Sebastien.

Merci à mes amis et à tout ceux qui ont participé de près ou de loin à l'accomplissement de ce travail.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

RÉSUMÉ

Malgré de nombreuses lacunes au niveau sécurité, les équipements sans-fil deviennent omniprésents : au travail, au café, à la maison, etc. Malheureusement, pour des raisons de convivialité, de simplicité ou par simple ignorance, ces équipements sont souvent configurés sans aucun service de sécurité, sinon un service minimal extrêmement vulnérable. Avec de telles configurations de base, plusieurs attaques sont facilement réalisables avec des moyens financiers négligeables et des connaissances techniques élémentaires. Les techniques de détection d'intrusions peuvent aider les administrateurs systèmes à détecter des comportements suspects et à prévenir les tentatives d'intrusions.

Nous avons modifié et étendu un outil existant (Orchids), basé sur la vérification de modèles, pour détecter des intrusions dans les réseaux sans-fil. Les attaques sont décrites de façon déclarative, à l'aide de signatures en logique temporelle. Nous avons tout d'abord développé et intégré, dans Orchids, notre propre module spécialisé dans l'analyse des événements survenant sur un réseau sans-fil 802.11. Par la suite, nous avons décrit, à l'aide de signatures, un certain nombre d'attaques, notamment, *ChopChop* — à notre connaissance, nous somme les premiers à détecter cette attaque —, *ARP Replay*, et la *deauthentication flooding*. Ces attaques ont ensuite été mises en oeuvre, puis détectées avec succès dans un environnement réel (trois machines: client, pirate et détecteur d'intrusion, plus un point d'accès).

Mots-clés : sécurité, détection d'intrusions, réseaux sans-fil, vérification de modèles.

INTRODUCTION

Malgré de nombreuses lacunes au niveau sécurité, les équipements sans-fil deviennent omni-présents : au travail, au café, à la maison, etc. C'est la simplicité de leur mise en place et de leur maintenance qui a amené à les préférer aux solutions filaires qui sont pourtant plus sécuritaires. Malheureusement, pour des raisons de convivialité, de simplicité ou par simple ignorance, ces équipements sont souvent configurés sans aucun service de sécurité, sinon un service minimal extrêmement vulnérable (WEP avec clé spécifique au constructeur).

Avec de telles configurations de base, plusieurs attaques sont facilement réalisables avec des moyens financiers négligeables et des connaissances techniques élémentaires. De ce fait, plusieurs solutions ont été proposées pour mieux sécuriser les réseaux sans-fil. Parmi ces solutions on trouve les corrections logicielles, les VPNs, les VLANs, les filtres, les détecteurs d'intrusions, etc. Chaque solution convient mieux à un besoin spécifique ou à une architecture spécifique. Par exemple, les techniques de détection d'intrusions peuvent aider les administrateurs systèmes à détecter des comportements suspects et à prévenir les tentatives d'intrusions.

Dans ce mémoire, on discute des solutions basées sur la détection d'intrusions dans les réseaux locaux sans-fil 802.11, et on propose une nouvelle approche fondée sur la logique décrite dans «*Log auditing through model-checking*» [26] et mise en œuvre dans Orchids [23].

Dans le premier chapitre, on fait une introduction à la technologie des réseaux locaux sans-fil 802.11. On s'intéresse particulièrement aux formats des trames et à la description des messages échangés pour le contrôle, la gestion, et l'exploitation des canaux de communication. On présente ensuite un aperçu de quelques faiblesses dans le standard IEEE 802.11 ainsi que les attaques qui permettent d'exploiter ces faiblesses.

Dans le deuxième chapitre, on présente plusieurs méthodes utilisées pour la détection des attaques et on montre que selon le besoin on peut préférer une méthode à une autre. On présente ensuite quelques défis auxquels sont confrontés les outils de détection d'intrusions. On termine le chapitre en présentant des solutions de détection d'intrusions spécifiques aux réseaux sans-fil 802.11.

Le troisième chapitre est consacré aux méthodes de vérification de modèles (*model-checking*). On s'intéressera plus particulièrement à la modélisation des systèmes réactifs à l'aide des automates et à la spécification des propriétés temporelles. Ce chapitre illustre, en fait, les fondements de la logique utilisée dans Orchids.

Dans le quatrième chapitre, on parle d'Orchids. On présente son architecture, la logique qu'il met en œuvre et son langage de spécification des attaques. On présente ensuite le processus d'analyse utilisé dans le noyau d'Orchids en se basant sur un exemple simple de simulation d'automate.

Ensuite, on présente successivement, dans les chapitres 5 et 6, les détails de la mise en œuvre du module sans-fil 802.11 pour Orchids et les expériences effectuées sur notre plateforme de tests ainsi que les résultats obtenus. Ces deux chapitres forment le noyau de notre propre contribution.

CHAPITRE I

RÉSEAUX LOCAUX SANS-FIL 802.11

Les technologies sans-fil gagnent du terrain par rapport au traditionnel câble et ce dans tous les domaines de l'informatique et des télécommunications. Que ce soit pour les longues distances avec le *WiMAX*, les courtes distances avec le *WiFi* ou les très courtes distances avec le *Bluetooth*, l'*infra-rouge*, etc., plusieurs raisons incitent à préférer le sans-fil au câble.

Dans ce chapitre, on va se limiter à la technologie 802.11[1] plus communément connue sous le nom commercial WiFi. Dans la première section, on introduit la terminologie utilisée dans ce domaine en présentant les protocoles associés et les différents modes de fonctionnement. Ensuite, pour une meilleure compréhension des techniques de détection d'intrusions illustrées dans la suite de ce mémoire, on a choisi de consacrer une section à la présentation des différentes catégories et en-têtes des trames 802.11. Enfin, on parlera des problèmes de sécurité observés avec cette technologie.

1.1 Introduction aux réseaux locaux sans-fil 802.11

Le premier standard de réseaux locaux sans-fil IEEE 802.11 a été créé en 1997. Cela dit, avec des débits de l'ordre de 1 à 2 Mégabits par seconde, ce standard n'a pas connu beaucoup de succès. Il a fallu attendre jusqu'en 1999 pour que la *Wi-Fi Alliance*[2] se mette à travailler avec l'IEEE pour pousser la vente de cette technologie en élaborant ce qui est devenu le premier standard de réseaux locaux sans-fil vendu dans des produits certifiés.

---

[1] Par abus de langage, on parle de «technologie 802.11» en faisant référence au standard 802.11 de l'IEEE *(Institute of Electrical and Electronic Engineers)* normalisant les technologies des réseaux locaux sans-fil.

[2] Organisation à but non lucratif regroupant plus de 250 membres incluant les plus grands industriels œuvrant dans le domaine du sans-fil (`www.wi-fi.org`).

Depuis, la norme IEEE 802.11 a évolué en proposant de meilleures solutions avec de plus hauts débits, une plus grande fiabilité, une plus grande couverture, une meilleure sécurité, etc. Plusieurs groupes de travail ont été créés au sein de l'IEEE pour s'attaquer à ces différentes problématiques[3]. Par exemple, le groupe TGn étudie la possibilité d'augmenter le débit à plus de 100 Mégabits par seconde. Le groupe TGr étudie, quant à lui, la possibilité de réduire le temps pour transférer la connexion d'une station lors de sa transition entre deux points d'accès (temps de *handover* horizontal). En ce qui concerne la problématique de la sécurité, le groupe TGi a fini par proposer le standard 802.11i améliorant les mécanismes de protection et d'authentification. De leur côté, les groupes TGe et ADS *(Advanced Security study group)* sont encore actifs sur cette même problématique en s'attaquant respectivement à la qualité de service et à la sécurité des trames de gestion.

La mise en œuvre des différents standards IEEE 802.11 est effectuée au niveau de trois couches : matériel, micrologiciel *(firmware)* et pilote (voir figure 1.1). La couche matériel est constituée essentiellement d'un jeu de puces *(chipset)* et d'une interface radio (une ou plusieurs antennes, un amplificateur, un synthétiseur, etc.). Le micrologiciel est embarqué dans le contrôleur d'accès au média *(Medium Access Controller)* situé au niveau du jeu de puces. C'est ce composant qui «accomplit la plus grande partie de la gestion fondamentale du protocole 802.11» [32]. Pour communiquer avec le matériel, le noyau du système d'exploitation a besoin d'un pilote qui sert aussi à gérer les interruptions. Le pilote est donc l'interlocuteur privilégié des deux côtés (système d'exploitation et matériel).

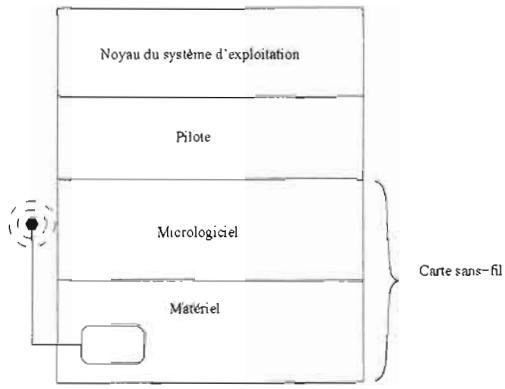

Figure 1.1 Couches impliquées dans la mise en œuvre des standards IEEE 802.11.

[3] `http://grouper.ieee.org/groups/802/11/QuickGuide_IEEE_802_WG_and_Activities.htm`

Plusieurs équipementiers ont choisi de publier la documentation de leurs *chipsets* (parfois partiellement) pour permettre ainsi à la communauté du logiciel libre de développer des pilotes à code source ouvert *(open source)*. D'autres ont, par contre, choisi de ne rien publier ; ceci n'a pas empêché des développeurs de faire un travail de *reverse engineering* et de distribuer des pilotes en logiciel libre (c'est le cas par exemple des interfaces à base du *chipset* ISL38xx d'Intersil[4]).

Chaque interface sans-fil (ensemble du matériel, du micrologiciel et du pilote) peut supporter un ou plusieurs modes de fonctionnement. Le mode ad hoc permet de créer une architecture réseau avec une cellule de base indépendante *(IBSS — Independent Basic Service Set)*. Dans ce cas, les stations peuvent communiquer soit directement, si elles sont assez proches, soit à travers d'autres stations (sans passer par un point d'accès). La figure 1.2 présente un exemple d'architecture avec IBSS. On voit que les stations STA3.2 et STA3.3 communiquent directement puisqu'elles sont physiquement proches. Par contre, STA2.3 doit passer par STA2.2 et STA1.2 pour arriver à communiquer avec STA1.1.

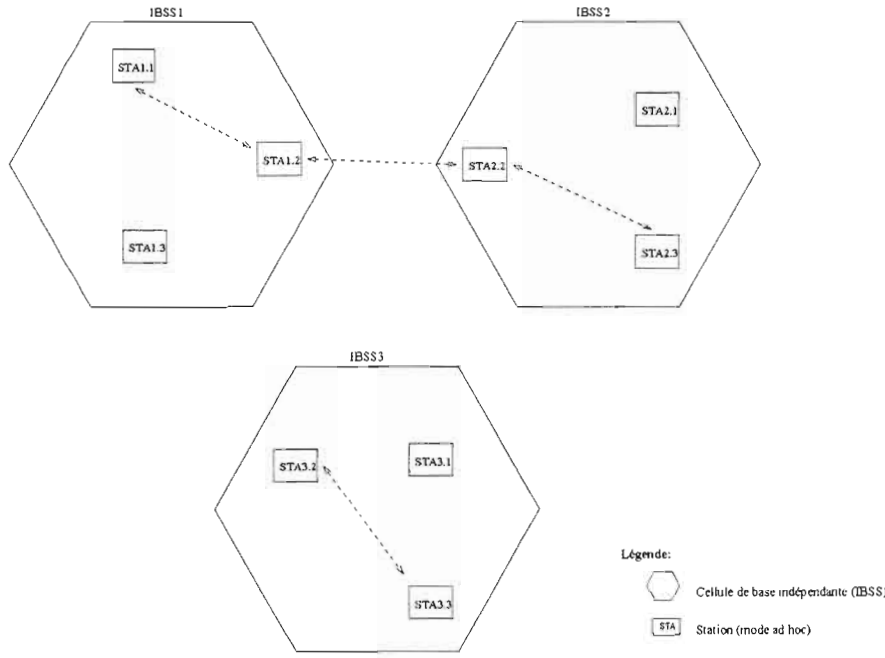

Figure 1.2 Architecture en mode ad hoc.

[4] www.intersil.com

Le mode maître *(master)* permet, quant à lui, à une interface 802.11 d'agir comme un point d'accès en offrant des services de synchronisation et de retransmission. Dans ce cas, on parle d'une architecture avec infrastructure dont le point d'accès représente l'élément central de chaque cellule de base *(BSS — Basic Service Set)*. Les stations doivent se mettre en mode contrôlé *(managed)* pour pouvoir utiliser les services d'un point d'accès. La figure 1.3 présente un exemple d'architecture avec infrastructure. On voit que chaque cellule de base contient un seul point d'accès et plusieurs stations. Les points d'accès peuvent communiquer entre eux en utilisant n'importe quelle technologie (filaire où sans-fil). Comme on peut le remarquer, bien que STA1.1 et STA1.2 soient proches l'une de l'autre, elles doivent impérativement passer par le point d'accès AP1 pour communiquer ; de même pour les stations STA2.2 et STA3.1.

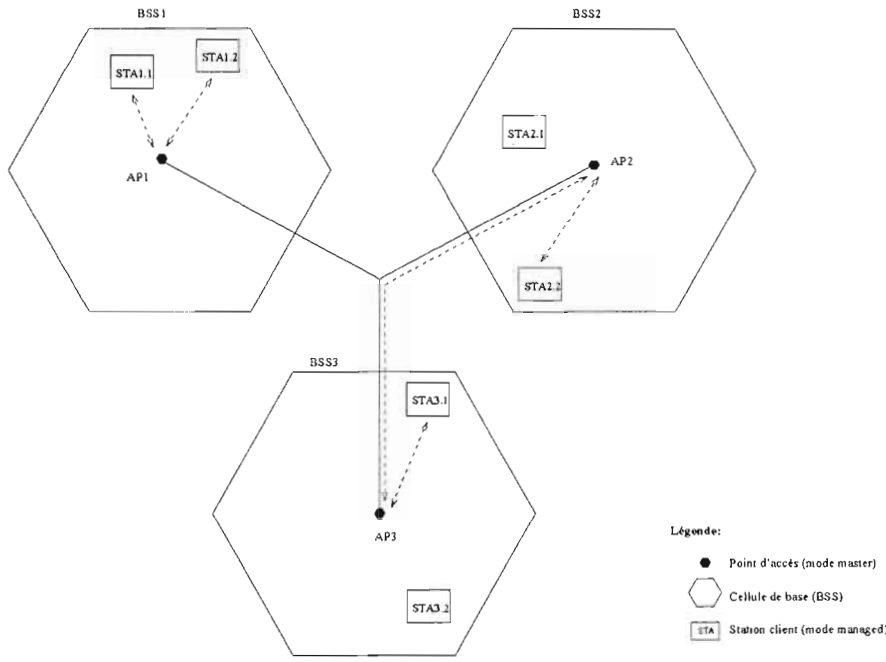

Figure 1.3 Architecture avec infrastructure.

Le mode moniteur *(RFMON — Radio Frequency Monitoring)* est souvent confondu avec le mode écoute *(promiscuous)* d'Ethernet, or ce sont des modes différents. En effet, si on prend l'exemple d'une architecture avec infrastructure présentée à la figure 1.4, on remarque que les zones de couverture des cellules de base se recouvrent en partie. Si la station STA1, qui se trouve dans un espace couvert exclusivement par la cellule de base BSS1, se met en mode moniteur, elle ne verra que les trames émises par le point d'accès AP1. Par contre, si la station STA2, qui

se trouve dans l'espace commun aux cellules de base BSS1 et BSS2, se met en mode moniteur, elle verra les trames émises par AP1 et AP2. Maintenant si STA2 se met en mode contrôlé, s'associe à AP1 et se met ensuite en mode écoute, elle ne verra que les trames émises par AP1. Donc, le mode écoute ne fait que désactiver le filtre d'adresses MAC *(Medium Access Control)* de la carte et permet ainsi d'accepter toutes les trames observées *sur le réseau auquel la carte est attachée*. Par contre, dans le mode moniteur, la carte n'est associée à aucun réseau et permet donc de récupérer toutes les trames qui l'atteignent.

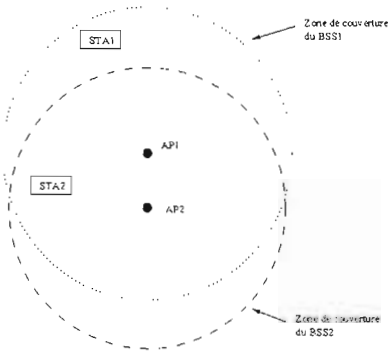

Figure 1.4 Exemple de deux cellules de base qui se chevauchent partiellement.

Il est important de noter que pour qu'une interface supporte un mode de fonctionnement spécifique, il faut non seulement que le *chipset* le supporte mais il faut aussi que le pilote le prenne en charge. Ainsi, il est possible de trouver une carte avec un *chipset* qui supporte le mode maître ou moniteur sans trouver le pilote qui les prend en charge.

Comme c'est le cas dans tous les protocoles de communication, l'échange de données s'effectue à la base par la transmission d'unités de données *(MPDU — Message Protocol Data Unit)*. Au niveau de la couche MAC *(Medium Access Control)* du 802.11, l'unité de données est la trame. Dans la section qui suit, on présente les différentes catégories et formats des trames 802.11.

1.2 Catégories et en-têtes des trames 802.11

Chaque trame 802.11 est composée d'un en-tête *(header)*, des données (contenant des informations spécifiques pour chaque type de trame) et d'une séquence de contrôle d'intégrité *(FCS — Frame Check Sequence)*. L'interprétation du format des trames dépend de leur type et de l'architecture considérée. Pour simplifier la présentation, nous allons nous limiter à l'architecture avec infrastructure.

La figure 1.5 représente le format général d'une trame 802.11 telle qu'elle est reçue par la couche MAC *(Medium Access Control)*. Dans ce qui suit, on présente l'interprétation des principaux champs qui nous intéressent.

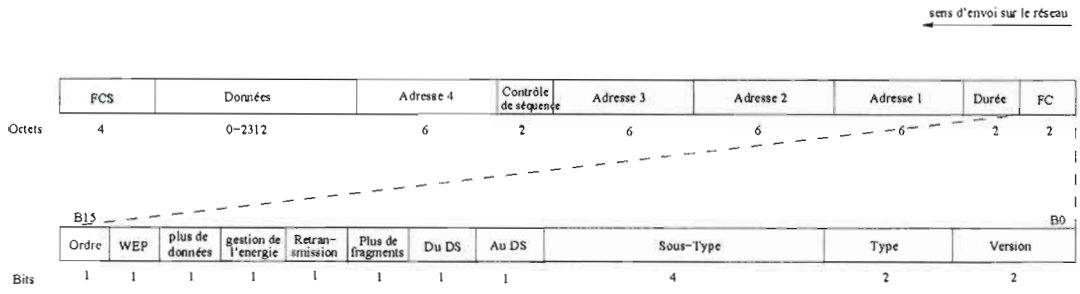

Figure 1.5 Format général d'une trame 802.11.

Le type de la trame est codé sur deux bits (champ Type dans la figure 1.5). Le standard 802.11 définit trois types de trames : les trames de contrôle permettant le contrôle du canal de communication (réservation, libération, etc.) et de l'energie, les trames de gestion permettant la gestion des communications (authentification, désauthentification, association, désassociation, etc.), et les trames de données. Chaque type de trame est associé à un ensemble de sous-types. Une combinaison valide de type et de sous-type permet d'identifier la fonction précise de la trame. Il faut noter ici qu'il y a des champs qui sont présents uniquement pour certains types de trames. Par exemple, le champ Données n'est pas présent dans les trames de gestion [16].

Les deux champs AuDS et DuDS indiquent la direction d'une trame de données. Ainsi, quand la valeur du champ AuDS vaut 1 et celle du champ DuDS vaut 0, on dira que la trame est envoyée par une station vers un point d'accès. Autrement, quand la valeur du champ AuDS vaut 0 et celle du champ DuDS vaut 1, on dira que la trame est envoyée par un point d'accès vers une station. Dans le cas où les deux champs sont à 0, cela signifie qu'il s'agit d'une trame provenant d'une station et destinée à une autre station. Finalement, si les deux champs sont à 1, il s'agit dans ce cas d'une trame envoyée entre deux points d'accès.

Les champs d'adresses MAC (Adresse1, Adresse2, Adresse3, Adresse4) tiennent sur six octets et indiquent — selon le type de la trame[5] et selon sa direction (champs AuDS et DuDS) — l'adresse source (adresse individuelle de l'émetteur de la trame), l'adresse destination (adresse individuelle ou de groupe identifiant la destination finale de la trame), l'adresse du point d'accès, l'adresse du point d'accès émetteur ou l'adresse du point d'accès récepteur.

[5] Certains types de trames ne contiennent que deux ou trois champs d'adresses.

Le champ ContrôleDeSéquence permet d'ordonner les trames dans le temps et de distinguer les divers fragments d'une même trame. Il se décompose en deux sous-champs : NuméroDeSéquence et NuméroDuFragment. Le NuméroDeSéquence permet d'identifier une trame dans un flot de données. Il commence avec la valeur 0 et est incrémenté avec chaque nouvelle trame transmise. Arrivé à la valeur 4096, le compteur recommence à compter à partir de 0.

1.3 Vulnérabilités dans les réseaux sans-fil 802.11

«Your 802.11 Wireless Network has no Clothes» [1], cette citation de William A. Arbaugh exprime bien l'extrême vulnérabilité des réseaux 802.11. En commençant par le niveau radioélectrique (couche physique) où il est toujours possible d'effectuer une attaque de déni de service à l'aide d'un brouilleur de fréquences et d'une antenne faits maison, pour arriver aux attaques les plus complexes ciblant des couches de plus haut niveau.

Bien qu'il soit possible de détecter une attaque de brouillage radio, dans cette section on se limite aux vulnérabilités dans la couche MAC 802.11 (couche 2 du modèle OSI). On se limitera aussi aux attaques actives car les attaques passives ne sont généralement pas détectables (quoiqu'on peut imaginer une technique de détection au niveau radioélectrique, ce qui sort du cadre de ce mémoire).

Une vulnérabilité est une faiblesse dans le système. Lorsqu'une telle faiblesse est exploitée, elle donne naissance à une attaque qui met en péril l'un des services de base de la sécurité. Dans ce qui suit, on a choisi de présenter les attaques en les classant par service de sécurité affecté.

1.3.1 Attaques contre l'authentification

Dans un réseau sans-fil 802.11, une station est identifiée à l'aide de son adresse MAC. Pour autoriser une station à s'associer au réseau, il faut vérifier son identité ; c'est ce qu'on appelle «authentifier la station». Le standard IEEE 802.11 propose plusieurs méthodes d'authentification, notamment via un système 802.1x ou à l'aide d'une clé partagée. Cela dit, dans une configuration par défaut, le système d'authentification est ouvert *(Open System authentication)* ; avec une telle configuration, l'authentification est effectuée à l'aide d'une simple notification.

Il est clair qu'en présence d'un système d'authentification ouvert, l'objectif d'«authentifier les stations» échoue. L'attaque d'usurpation d'adresse MAC *(MAC spoofing)* consiste donc à se faire passer pour quelqu'un qu'on n'est pas en réalité. Il suffit à l'intrus d'utiliser l'identité d'une

autre station (son adresse MAC) soit pour monter une attaque sans se faire repérer, soit pour accéder à des services privilégiés, soit pour détourner un système de filtrage par adresse MAC. Techniquement, il est facile de changer l'adresse MAC d'une interface sans-fil. Mais l'intrus n'a pas besoin de changer son adresse puisqu'il lui suffit de forger une trame 802.11 avec les adresses MAC de son choix.

Plusieurs travaux ont tenté de détecter l'attaque *MAC spoofing* [5; 12; 34] mais aucun ne réussit à le faire de façon efficace — des cas de faux positifs et de faux négatifs persistent. Cette attaque est intéressante car elle est généralement la première brique sur laquelle sont construites la plupart des attaques dans les réseaux sans-fil [2; 5; 20; 28]. Autrement dit, si on arrivait à détecter efficacement l'attaque d'usurpation d'adresse MAC, on éliminerait un bon nombre d'attaques (connues ou inconnues) qui sont basées sur cette technique.

1.3.2 Attaques contre la disponibilité (déni de service)

Talon d'achille des technologies sans-fil, les attaques de déni de service sont les plus simples et les plus faciles à effectuer. Le but étant d'interrompre un service soit de façon permanente soit de façon temporaire (seulement pour gêner les utilisateurs du service), l'intrus dispose d'une panoplie d'outils qui exploitent plusieurs faiblesses dans le système.

Une des attaques les plus populaires[6] est l'attaque par inondation de trames de type désauthentification (*deauthentication flooding*). Cette attaque est possible du fait que le protocole n'authentifie pas les trames de gestion, ce qui facilite leur usurpation (*spoofing*).

Le standard IEEE 802.11 a prévu la possibilité de désauthentifier une station, soit de sa propre initiative lorsque l'utilisateur n'a plus besoin du réseau, soit par le point d'accès lorsqu'il lui faut une mise à niveau (après une opération de maintenance de l'équipement) ou pour toute autre raison. Dans les deux cas, une simple notification est envoyée à l'entité correspondante (à la station mobile quand la consigne vient du point d'accès, et au point d'accès quand la consigne vient de la station mobile). Il suffit donc à un intrus de se faire passer soit pour le point d'accès, soit pour la station en envoyant une trame de type désauthentification forgée avec les adresses MAC source et destination respectives (voir figure 1.6). Suite à la réception de cette trame, aucune communication n'est possible avant une ré-authentification de la station. Si l'intrus persiste en envoyant continuellement (*flooding*) des trames de ce type, la station serait dans ce cas indéfiniment déconnectée du réseau.

---

[6] Malgré sa popularité elle est encore réalisable.

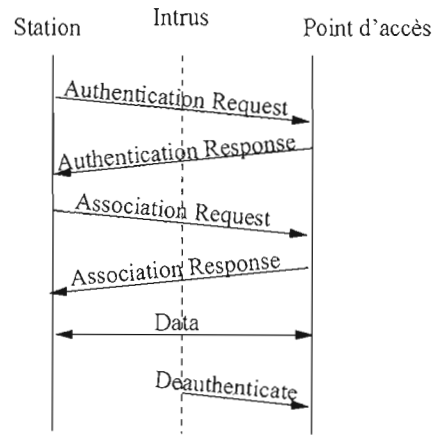

Figure 1.6 Attaque de désauthentification (figure adaptée de [2]).

Le problème avec cette attaque, c'est qu'elle est très flexible. L'intrus peut se contenter de gêner un utilisateur en amenant sa station à se ré-authentifier à une fréquence donnée (il limite ainsi sa connectivité et son débit), comme il peut carrément bloquer l'accès au canal de communication à toutes les stations.

Une autre attaque assez semblable est l'attaque de désassociation (*disassociate flooding*). Même principe et même cause, parce que les trames de désassociation ne sont pas authentifiées, donc quiconque peut les usurper. La seule différence est qu'en se dissociant, la station d'un utilisateur ne perd pas son authentification et donc il lui suffit de se ré-associer pour rétablir la communication. Ceci rend cette attaque un peu moins efficace par rapport à l'attaque de désauthentification puisque la station doit échanger un nombre réduit de messages avant de pouvoir rétablir la communication.

Toujours dans la même catégorie, on trouve l'attaque par inondation RTS *(Request To Send)* fondée sur l'exploitation du mécanisme de réservation. L'intrus est capable de bloquer le canal de communication en envoyant des trames de réservation (RTS) avec un champ durée fixé sur un grand intervalle. On se trouve donc dans une situation où le canal reste monopolisé par l'intrus qui n'arrête pas de faire des réservations successives. Cependant, Bellardo et Savage ont constaté qu'en pratique on peut trouver des implémentations qui ne respectent pas le système de réservation du standard 802.11 et qui sont par conséquent insensibles à cette attaque [2]. Cela ne veut pas dire que ces implémentations sont protégées contre cette attaque, mais plutôt qu'elles contiennent des incohérences par rapport au standard et qu'elles doivent être corrigées.

Une autre attaque décrite par Bellardo et Savage est l'attaque contre le système de gestion de l'énergie [2]. Pour conserver de l'énergie, une station peut entrer en mode repos durant lequel elle ne peut ni envoyer ni recevoir des données. Le point d'accès stocke dans ce cas tout le trafic qui lui est destiné. Ensuite, la station se réveille occasionnellement pour récupérer (à sa demande) les trames en attente et autorise ainsi le point d'accès à libérer son tampon. Si un intrus se fait passer pour la station pendant que celle-ci est en mode repos, et qu'il demande de récupérer les trames en attente, ceci provoque la suppression des données mises dans le tampon par le point d'accès.

1.3.3 Attaques contre la confidentialité

Évidemment, si on n'utilise aucun mécanisme de chiffrement, on court le risque que nos données puissent être facilement interceptées par une personne malveillante. Le standard IEEE 802.11 définit trois mécanismes de chiffrement : WEP *(Wired Equivalent Privacy)*, TKIP *(Temporal Key Integrity Protocol)*, et CCMP *(Counter Mode with Cipher Block Chaining Message Authentication Code Protocol)*. Cela dit, la plupart des équipements sans-fil sont commercialisés avec une configuration par défaut qui ne procure aucune protection. L'utilisateur doit donc configurer son équipement pour l'amener à exploiter l'un des mécanismes de chiffrement cités plus haut.

Le WEP est le plus simple à paramétrer, mais il est aussi le plus vulnérable. En effet, malgré qu'«il n'existe pas à l'heure actuelle de technique connue permettant de casser les protocoles TKIP et CCMP» [5], et à cause de la nécessité d'un déploiment laborieux pour ces deux protocoles, la plupart des équipements sans-fil sont activés soit sans aucun mécanisme de chiffrement soit avec le WEP. Donc, comme le fait remarquer Vladimirov *et al.*, le WEP restera d'usage pour encore un bon moment indépendamment du niveau de sécurité proposé par ses successeurs potentiels [32], notamment grâce à la simplicité de sa mise en place.

La figure 1.7 illustre le fonctionnement du WEP du côté de l'émetteur (mode chiffrement). La première étape consiste à appliquer une fonction CRC-32 sur le texte en clair pour générer un ICV *(Integrity Check Value)* qui servira plus tard au récepteur pour vérifier l'intégrité du message reçu. D'un autre côté, on produit un flux de chiffrement en applicant une fonction RC4 *(Ron's Code 4)*[7] sur la clé WEP composée par un vecteur d'initialisation *(IV — Initialization Vector)*

[7] RC4 a été développé par Ronald L. Rivest et est une marque enregistrée de l'entreprise *RSA Security*

et une clé pré-partagée *(PSK — Pre-Shared Key)*. Ensuite, pour produire le message chiffré, il suffit d'effectuer un ou-exclusif (opérateur XOR) entre le flux de chiffrement et l'ensemble composé par le texte en clair et l'ICV. Finalement, on compose la trame 802.11 en préfixant le message chiffré par le vecteur d'initialisation et par l'en-tête MAC, et en le suffixant par le FCS *(Frame Check Sequence)*. Remarquez que le vecteur d'initialisation est transmis en clair pour permettre au récepteur de générer le même flux qui a servi au chiffrement.

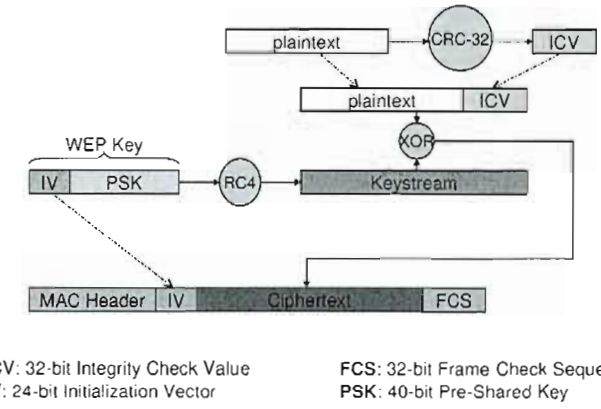

Figure 1.7 Fonctionnement du WEP en mode chiffrement (figure inspirée de [16]).

Donc, à la réception de la trame 802.11, le récepteur la découpe, extrait le vecteur d'initialisation et le concatène avec la clé pré-partagée pour former la clé WEP, applique la fonction RC4 sur cette clé pour produire le flux de déchiffrement, effectue un ou-exclusif entre le flux de déchiffrement et le message chiffré pour obtenir le texte en clair et l'ICV associé, et applique finalement la fonction CRC-32 sur le texte en clair pour s'assurer que le message n'a pas été modifié depuis sa transmission.

Les faiblesses du WEP ont fait l'objet d'innombrables publications [30; 4], la plus évidente concerne le fait que la fonction CRC-32 a été inventée initialement dans le but de détecter des erreurs de transmission et non pour assurer l'objectif plus exigeant de contrôle d'intégrité. Plusieurs attaques[8] ont exploité l'une ou plusieurs de ces faiblesses pour casser le WEP : *dwepcrack*, *wepAttack*, *wep-tools*, *wep-crack*, *AirSnort*, *ChopChop*, *ARP Reply*, etc. On a choisi

---

Inc. (www.rsa.com).

[8] www.wi-foo.com

de présenter ici l'attaque ChopChop qu'on va décrire à l'aide du langage de signature d'Orchids dans le chapitre 5 «Mise en œuvre du module 802.11 et des signatures d'attaques».

ChopChop est le nom de l'outil publié par KoreK (un pseudonyme) le 14 septembre 2004 sur un forum de discussion du site `NetStumbler.org`. Cette attaque exploite une vulnérabilité liée à l'utilisation de la fonction CRC-32 dans WEP.

La figure 1.8 illustre le déroulement de l'attaque ChopChop. On commence par supposer que l'octet de poids faible du texte en clair est un zéro (les huit bits à zéro). On enlève ensuite la valeur chiffrée de cet octet dans le message chiffré et on obtient donc un nouveau message chiffré malformé car l'ICV est celui qui a été obtenu par le calcul de la fonction CRC-32 sur le message initial entier (message M) au lieu du message réduit d'un octet (message M-1). Pour corriger cette malformation, KoreK a proposé une fonction mathématique permettant d'enlever l'effet du zéro sur l'ICV. Le message obtenu suite à l'application de cette fonction devrait être valide seulement si notre supposition initiale est bonne (l'octet de poids faible du texte en clair est un zéro). Pour vérifier si on a bien deviné, on envoie le nouveau message chiffré à un point d'accès. Si le message est bon il est retransmis par le point d'accès, sinon il est rejeté.

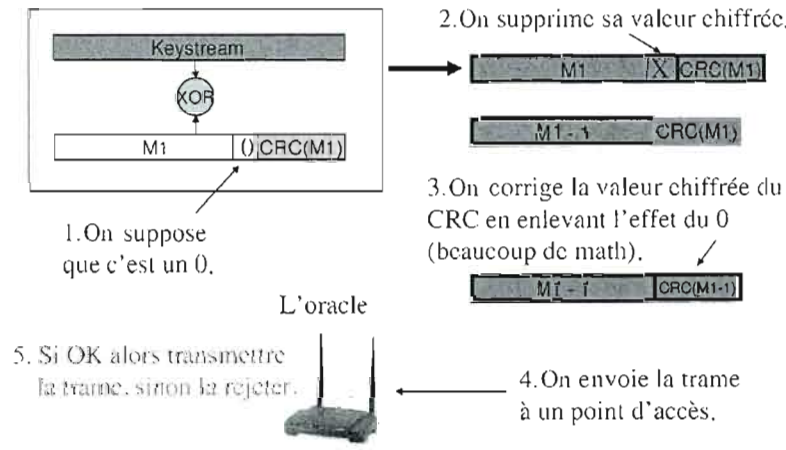

Figure 1.8 Déroulement de l'attaque ChopChop.

Dans le cas où le message est rejeté, on refait les mêmes opérations en incrémentant de 1 la valeur supposée pour le dernier octet, ceci jusqu'à ce qu'on trouve la bonne valeur. Quand on réussit à deviner le dernier octet, on l'enlève du message et on refait la même chose avec l'octet suivant jusqu'à ce qu'on découvre tout le message en clair, un octet à la fois.

Pour détecter les différentes attaques illustrées dans cette section, on utilise ce qu'on appelle «un détecteur d'intrusions». On verra dans le chapitre suivant quelques méthodes et approches utilisées pour la détection d'intrusions ainsi que les défis auxquels elles sont confrontées. On notera aussi la particularité de la détection d'intrusions dans les réseaux sans-fil 802.11 par rapport à celle effectuée dans les réseaux filaires.

CHAPITRE II

DÉTECTION D'INTRUSIONS

À cause de leur grande complexité et de leur besoin d'ouverture sur l'environnement extérieur, les systèmes ouverts peuvent permettre la réalisation d'opérations imprévues (bénignes ou malignes). Une opération maligne est tout enchaînement d'actions élémentaires conduisant à une violation de la politique de sécurité et on considère cette possibilité comme une faiblesse (vulnérabilité) dans le système. On appelle les opérations malignes des *attaques* et les opérations bénignes des *features*. Par abus de langage on appelle l'outil effectuant la détection des attaques «un détecteur d'intrusions».

On peut penser, à tort, que la détection d'intrusions est inutile et qu'en faisant le parallèle avec le marché des anti-virus, c'est une raison de plus pour nous vendre des outils en nous faisant peur avec des noms d'attaques exotiques comme : *teardrop, ping of death, smurf, bonk, brkill, oshare, coke,* etc.[1]

En fait, la durée entre le moment de la publication d'une vulnérabilité, la publication du correctif et l'application de ce dernier peut être considérable. Il suffit de consulter la base des vulnérabilités classées par le CERT pour s'en rendre compte. Par exemple, la vulnérabilité de quelques systèmes *Microsoft Windows* face à l'attaque LAND a été publiée la première fois par Dejan Levaja le 5 mars 2005 [2] [3]. Il a fallu attendre jusqu'au 12 avril 2005 pour voir le premier correctif publié par *Microsoft* (MS05-019[4]) — et un autre mois pour le correctif du correctif.

---

[1] `http://www.securiteinfo.com/sommaire_hacking.shtml`.

[2] *Vulnerability Note* VU#396645 : `http://www.kb.cert.org/vuls/id/396645`.

[3] `http://www.securityfocus.com/archive/1/392354`.

[4] `http://www.microsoft.com/technet/security/bulletin/ms05-019.mspx`.

Durant tout ce temps, le système vulnérable n'a probablement aucun moyen de défense, d'où le besoin d'un outil qui permet de détecter les attaques exploitant cette vulnérabilité et qui pourrait même les contrer. Mais, même si on se trouve avec un système mis à jour et pratiquement invulnérable, la détection des *tentatives* d'intrusions est aussi intéressante dans certains cas — même si la dite intrusion échoue.

Connaître les particularités d'un outil de détection d'intrusions par rapport à un autre est essentiel si on veut tirer le meilleur de chacun. Plus généralement, pour un système spécifique (ou un besoin spécifique), une méthode de détection pourrait s'avérer plus appropriée (plus intéressante) qu'une autre. Par exemple, dans un réseau où le débit est faible, on n'a pas besoin d'une méthode d'analyse rapide puisque la quantité de données à analyser n'est pas aussi élevée que dans un réseau haut débit. De la même façon, les réseaux sans-fil ont des spécificités qu'il faut prendre en considération.

Dans la section suivante, on commence par faire ressortir les principaux critères de classification des détecteurs d'intrusions. Ensuite, on illustre quelques méthodes d'analyse usuelles, et on discute des nouveaux défis lancés aux détecteurs d'intrusions. Pour finir, on présente quelques outils de détection d'intrusions dans les réseaux sans-fil 802.11.

2.1 Classification des détecteurs d'intrusions

Plusieurs critères nous permettent de classer les détecteurs d'intrusions. On présentera dans ce qui suit les principales catégories qu'on retrouve dans la littérature.

2.1.1 Classification par source d'informations

Les détecteurs d'intrusions se basent sur des événements. Un événement est le résultat d'une action élémentaire sur le système. Un événement est souvent matérialisé sous forme d'une ligne dans un fichier de journalisation ou sous forme d'une trame (ou fragment d'une trame) transmise sur le réseau ou sous toute autre forme logicielle. On peut classer les événements en trois catégories : selon qu'ils proviennent du réseau, des systèmes, ou des applications.

Les sondes réseau *(NIDS — Network-based IDS)* récoltent tout événement matérialisé sur le réseau. Ces événements sont soit à l'état brut (non encore interprétés), soit qu'ils ont déjà traversé quelques nœuds (routeurs, commutateurs, pare-feu, points d'accès, etc.) avant d'arriver à la sonde. L'avantage avec ce type de sondes est que l'analyse peut être effectuée sur un système

dédié sans affecter les performances du réseau surveillé. Cependant, avec l'augmentation des débits (on parle aujourd'hui de quelques Gigabits par seconde) l'analyse en temps réel devient de plus en plus difficile à cause du grand volume de données récoltées. L'autre complication pour les sondes réseau est l'utilisation des protocoles cryptographiques, ce qui empêche l'analyse du contenu (voire des en-têtes).

Les sondes systèmes *(HIDS — Host-based IDS)*, quant à elles, se greffent sur les systèmes surveillés et récoltent seulement les événements matérialisés sur ces systèmes. Ces événements proviennent principalement du noyau et des modules du système d'exploitation. L'inconvénient avec ce type de sondes est que les performances des systèmes surveillés sont affectées (ce qui n'est pas le cas avec des sondes réseau). Cette perte en performances est le prix à payer pour une analyse plus précise et plus fine sur les appels systèmes du noyau. En plus, on n'est plus confronté au problème des données chiffrées auquel font face les sondes réseau puisqu'au niveau du système les événements ne sont pas chiffrés.

Finalement, les sondes applicatives *(AIDS — Application-based IDS)* surveillent les journaux spécifiques des applications (surtout ceux des serveurs comme les serveurs Web, ftp, etc.) indépendamment du système sur lequel elles se trouvent. L'avantage est qu'on est à un niveau où plusieurs événements élémentaires sont regroupés ensemble pour former «un plus gros» événement qui est sémantiquement plus riche. Morin a donné un bon exemple pour illustrer cette idée. Il s'agit de l'exemple d'une réponse à une requête HTTP qui donne lieu à une séquence de plusieurs paquets IP au niveau du réseau, alors qu'elle ne donne lieu qu'à une seule ligne dans le fichier d'audit du serveur Web [19].

2.1.2 Classification par approche de détection

On peut classer les méthodes de détection en deux catégories. Dans la première catégorie, on trouve les méthodes d'analyse qui s'intéressent aux actions interdites dans le système surveillé ; on parlera alors d'une approche par détection d'abus *(misuse detection)* ou approche par scénarios. Quant à la deuxième catégorie, on trouve les méthodes d'analyse qui s'intéressent aux actions autorisées et on parlera donc d'une approche par détection d'anomalies *(anomaly detection)* ou approche comportementale. Ces deux approches sont opposées mais complémentaires.

Avec l'approche par détection d'abus *(misuse detection)*, on dispose d'une banque de signatures d'attaques (actions illégales, voir figure 2.1) servant à identifier des attaques. On propose donc de chercher les occurrences de ces signatures dans un flux d'événements ; si on trouve une concordance, on lance une alerte. «Ainsi, tout ce qui n'est pas explicitement défini est autorisé» [19] et tout ce qui est explicitement défini est interdit.

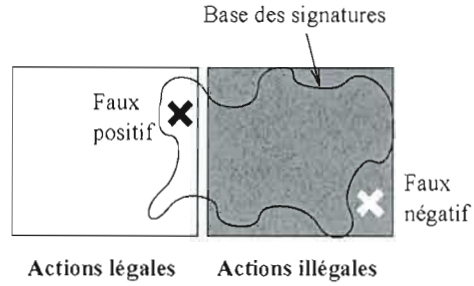

Figure 2.1 Approche par scénarios [3].

L'inconvénient avec cette approche est qu'elle permet uniquement la détection d'attaques connues. Il faut donc tenir à jour la base des signatures sinon on risque d'avoir plusieurs faux négatifs (voir figure 2.1) — «Une concordance fausse négative signifie qu'une concordance qui aurait due être réussie ne l'a pas été» [24]. En plus, la complexité du système surveillé pourrait rendre difficile la distinction entre des actions légales et des actions illégales, ce qui risque de produire de fausses alertes — «Une concordance fausse positive signifie qu'on propose une concordance, mais qu'en réalité ce n'en est pas une» [24]. En revanche, cette approche a le mérite d'être une approche simple, basée sur des algorithmes performants (en temps de calcul ainsi qu'en consommation de mémoire).

L'approche par détection d'anomalies *(anomaly detection)*, appelée aussi approche comportementale, consiste à définir ce qu'on considère comme comportement *normal* (actions légales, voir figure 2.2). Ainsi, tout événement qui viole cette définition (ce modèle) est considéré comme suspect. «*A contrario* de l'approche précédente, tout ce qui n'est pas explicitement défini est interdit» [19] et tout ce qui est explicitement défini est autorisé.

Le principal avantage avec cette approche est qu'elle permet la détection d'attaques inconnues. Mais, d'un autre côté, elle est sujette à plusieurs faux positifs si le modèle comportemental n'est pas complet (ou s'il est trop restrictif, voir figure 2.2). De plus, si ce modèle est construit à l'aide d'un algorithme d'apprentissage, et que durant la phase d'apprentissage un comportement malveillant se produit, ce dernier sera automatiquement inclus dans le modèle et ne déclenchera pas d'alerte durant la phase de recherche d'attaque (faux négatif).

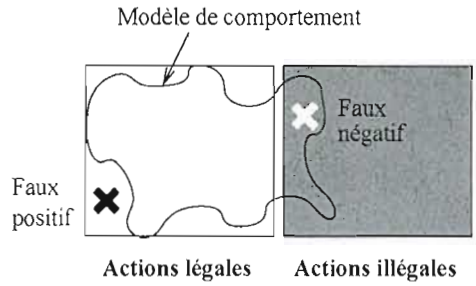

Figure 2.2 Approche comportementale [3].

2.1.3 Classification par mode d'utilisation

Selon les besoins, on peut choisir le mode d'utilisation d'un détecteur d'intrusions. Par exemple, dans le cadre d'un audit effectué par un organisme externe à une entreprise, une analyse des anciens journaux pourrait être pertinente pour évaluer l'efficacité d'une solution de détection d'intrusions mise en oeuvre à l'intérieur de cette même entreprise. Dans ce cas, l'équipe qui effectue l'audit a besoin d'un outil de détection d'intrusions qui sera utilisé occasionnellement. Par contre, l'entreprise auditée a besoin d'un outil qui est capable d'effectuer des analyses en temps réel. Chaque type d'analyse a dés avantages et des inconvénients.

L'analyse en temps réel (appelée aussi *on the fly* ou *online*) est une analyse effectuée «au fur et à mesure de la réception des événements» [22]. Le principal avantage est que les alertes sont lancées dès que les attaques sont détectées. Cet état de veille coûte cher en termes de ressources et nécessite des algorithmes plus complexes que ceux d'une analyse en différé.

L'analyse en différé *(offline)* est une analyse effectuée sur des données stockées (non fraîches). Vu le caractère non-pressant de ce type d'analyse (à tête reposée), les algorithmes sont plus simples. Sauf que la détection des attaques se fait après coup et nécessite le stockage des événements avant leur analyse.

2.1.4 Classification par type de réaction

De toute évidence, un détecteur d'intrusions qui ne réagit pas ne sert pas à grand chose. Le minimum que doit assurer un tel outil lorsqu'il détecte une attaque est de consigner cette information dans un journal. Après, si on a plus d'ambition, on peut espérer qu'il riposte aux attaques et qu'il identifie et localise de façon précise et complète l'intrus. On distingue alors deux types de détecteurs d'intrusions : passifs et actifs.

Les détecteurs d'intrusions passifs le sont seulement par rapport aux attaques. Cela dit, ils peuvent effectuer plusieurs opérations de façon active comme récolter des informations sur l'attaquant, alerter l'administrateur en lui envoyant un courrier électronique ou un SMS, produire un rapport détaillé sur les attaques détectées, etc.

Les détecteurs d'intrusions actifs ont pour objectif d'empêcher (dans la mesure du possible) le succès d'une attaque ou du moins de limiter son impact. Le terme IPS *(Intrusion Prevention Systems)* est de plus en plus utilisé, dans la presse spécialisée et par les services commerciaux des compagnies œuvrant dans ce domaine, pour designer les détecteurs d'intrusions actifs.

Par opposition aux détecteurs d'intrusions passifs, les détecteurs actifs (ou encore les IPS si on veut suivre la tendance) peuvent changer la configuration des systèmes ou du réseau. Ainsi, les réponses peuvent aller de la simple déconnexion de l'intrus, l'arrêt d'un processus, le verrouillage d'un compte utilisateur, jusqu'au changement des règles de filtrage d'un pare-feu ou même des règles de routage d'un routeur. Pour plus d'efficacité, les IPS sont généralement positionnés en coupure dans l'architecture du réseau (comme un pare-feu). Snort-Inline est un exemple d'IPS.

2.2 Méthodes d'analyse

«Les méthodes d'analyse représentent les techniques utilisées pour exploiter les informations récoltées» [22]. Il ne faut pas les confondre avec les approches de détection vues plus haut (section 2.1.2). En fait, les méthodes d'analyse mettent en œuvre l'une des deux approches : par scénarios ou comportementale. Dans les paragraphes suivants, on présente quelques méthodes d'analyse.

Méthodes statistiques

Cette méthode consiste en un calcul de la variation (en fonction du temps) de l'utilisation d'un certain nombre de ressources, à l'aide d'outils statistiques. La constatation d'un changement dans la variation conduit au déclenchement d'une alerte. Par exemple, «un trafic réseau anormalement faible en matinée signale très probablement une panne ou une attaque de type déni de service» [35].

Vérification de protocoles

Avec cette méthode, on ne cherche pas une attaque précise et connue mais on cherche une incohérence dans le déroulement d'un protocole ou dans les champs des en-têtes. Par exemple, dans l'attaque LAND on retrouve dans un même paquet TCP/IP la même valeur dans les champs correspondant aux ports source et destination et la même valeur dans les champs correspondant aux adresses IP source et destination, ce qui est inhabituel. En effet, plusieurs attaques «exploitent des cas imprévus par certains protocoles» [22] ou par leur mise en œuvre.

Recherche de motifs *(Pattern matching)*

Cette méthode consiste à chercher un motif (une séquence de caractères ou de bits) dans un flux d'événements. Le motif est généralement exprimé à l'aide d'un langage spécifique tel que les expressions régulières. Il faut aussi un algorithme efficace pour effectuer la recherche le plus rapidement possible.

Analyse d'automates

«Cette méthode consiste à baser la détection sur la simulation de modèles sous forme d'automates finis» [22] (c'est la méthode utilisée dans Orchids). Les signatures d'attaques sont, dans ce cas, vues à un niveau abstrait comme des automates finis.

Chaque règle décrit un comportement (séquence d'événements) qu'on veut détecter. On peut soit décrire un comportement malicieux pour détecter une attaque (approche par scénarios), soit décrire le comportement «normal» pour détecter un comportement «inhabituel» (approche comportementale). L'objectif dans tous les cas est de déclencher une alerte suite à l'observation d'une séquence suspecte d'événements.

2.3 Défis de la détection d'intrusions

Dans ce qui suit, on présente les principaux problèmes auxquels font face les détecteurs d'intrusions.

2.3.1 Prise en charge de l'aspect temporel

Comme l'indique Julien Olivain [22] :

> L'un des principaux problèmes dont souffrent les systèmes de détection, est effectivement l'aspect temporel dans les spécifications logiques des attaques ou des comportements, selon l'approche d'analyse choisie. La plupart du temps, les règles de détection d'intrusions définissent des contraintes sur des contenus indépendamment du temps.

Il est clair que la corrélation des événements dans le temps, en vue de la détection d'intrusions, serait handicapée en présence d'horloges désynchronisées. Pour illustrer cette idée, on prend un exemple simple d'une signature d'attaque définie par la succession de deux événements A et B matérialisés sur deux systèmes différents. En d'autres termes, on voudrait lancer une alerte si un système S1 engendre un événement A à un instant t et qu'un autre système S2 engendre un événement B à un instant $t + \Delta t$.

Dans le cas où les deux systèmes S1 et S2 sont parfaitement synchronisés, même si le détecteur d'intrusions observe l'événement B avant d'observer l'événement A (suite à un retard provoqué par une congestion dans un nœud du réseau ou pour toute autre raison), il est toujours possible de corréler les deux événements sur la base de leur datation.

Par contre, dans le cas où les horloges des deux systèmes sont désynchronisées, même si le détecteur d'intrusions observe A et tout de suite après observe B, il ne peut pas être certain du séquencement des deux événements (le problème est accentué avec la présence des retards dûs au temps de traitement).

D'un autre côté, «les différents systèmes de synchronisation d'horloges offrent des précisions et des garanties de synchronisation différentes» [22]. La figure 2.3 présente, par exemple, les désynchronisations des horloges de machines synchronisées par NTP *(Network Time Protocol)*. En haut à droite, on trouve une liste de noms d'ordinateurs : cassis, airelle, tamarin, tomate, etc. À chaque nom est associé un type de trait avec lequel est dessinée la courbe de désynchronisation de l'horloge de l'ordinateur correspondant.

Une horloge parfaitement synchrone devrait correspondre à l'axe des abscisses (la désynchronisation est égale à zéro). Plus la courbe s'éloigne de l'axe des abscisses (vers le haut ou vers le bas) plus la valeur du décalage est importante. Sur la figure 2.3, on constate des décalages allant jusqu'à plusieurs millisecondes (ce qui est énorme). Ceci voudrait dire qu'on ne peut pas corréler des événements provenant d'endroits différents et ayant des différences de dates au-dessous de ce seuil de précision.

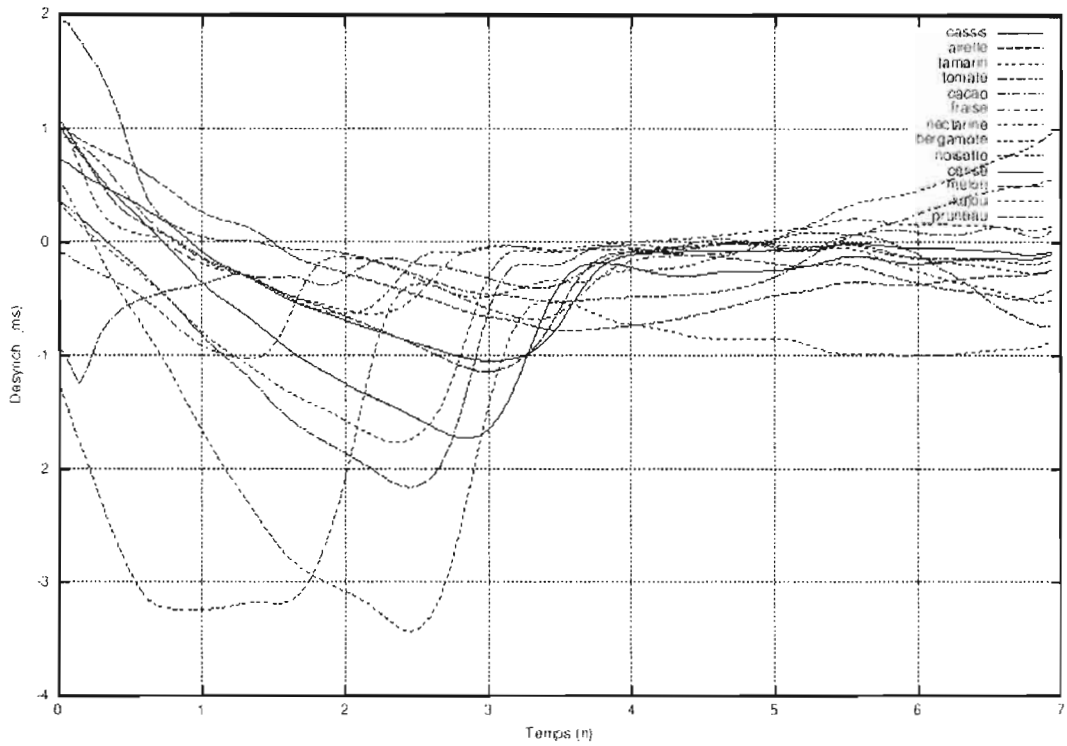

Figure 2.3 Désynchronisation NTP *(Network Time Protocol)* [22].

2.3.2 Multiplication des alertes

La multiplication des alertes est l'un des principaux problèmes des détecteurs d'intrusions. En effet, si un administrateur du système informatique se trouve noyé dans des journaux d'alertes et qu'en plus il constate que les alertes signalées ne sont pas toutes pertinentes et utiles, il finira par ne plus prendre au sérieux l'outil de détection d'intrusions.

La multiplication des alertes peut être la conséquence d'une ou de plusieurs caractéristiques des attaques, de l'architecture de la solution de détection ou encore de la méthode d'analyse. Benjamin Morin a défini trois principales sources des multiplications [19] : la récurrence, la redondance et la division.

En effet, une même attaque répétée durant un intervalle de temps défini ne doit pas engendrer des alertes récurrentes. Par exemple, une attaque de *scan* de ports ou de *SYN flooding* peut durer un bon moment ; et si durant ce temps le détecteur d'intrusions n'arrête pas de lancer des alertes par rapport à ces attaques, on va se retrouver avec des journaux inutilement remplis.

En plus, il est parfois intéressant de placer plusieurs sondes (du même type ou de types différents) sur un même système (ou réseau) surtout pour un maximum de couverture et pour arriver à traiter la totalité des événements produits. Cette multiplication des sondes implique une redondance des alertes : une même attaque constatée par plus d'une sonde engendre plusieurs alertes.

D'un autre côté, il n'est pas impossible qu'un même événement puisse concorder avec plus d'une signature d'attaque lorsqu'il «présente plusieurs propriétés suspectes» [19]. Benjamin Morin a présenté dans sa thèse de doctorat [19] l'exemple de la requête HTTP suivante pour expliquer cette idée :

```
http://webserveur/cgi-bin/phf?Qalias=x%0a/bin/cat%20/etc/passwd
```

Cette requête représente une attaque exploitant la vulnérabilité PHF *(Phone book HTML Form)*[5] en permettant à un attaquant d'afficher le contenu du fichier /etc/passwd du système

[5]PHF est le nom d'un script CGI intégré auparavant aux serveurs Web NCSA *(National Center for Supercomputing Applications)* et Apache. Le script permettait normalement la gestion des contacts téléphoniques.

vulnérable. «Certains IDS génèrent deux alertes pour cette requête : l'une concernant la présence de la chaîne de caractères phf? et l'autre concernant la présence de la chaîne /etc/passwd» [19] — pourtant, il s'agit bien d'une seule attaque.

2.3.3 Expressivité des langages de signatures

Pour exprimer une signature d'attaque ou un comportement il faut choisir un langage d'expression abstrait qui soit le plus proche de la réalité mais aussi le plus simple (qui fait donc abstraction de la réalité) pour une analyse plus rapide. Le problème est que, d'un côté, trop d'abstraction par rapport à la réalité limite l'expression et favorise l'ambiguité ; d'un autre côté, plus on s'approche du modèle réel plus on ralentit les algorithmes d'analyse (à cause de la complexité des structures).

Pour expliquer cette idée, prenons un exemple où on veut exprimer des émotions de façon graphique. On pourrait considérer dans ce cas, les yeux comme un langage abstrait d'expression des émotions sur un visage. Cela dit, ce langage fait abstraction d'une grande partie du visage (bouche, nez, rides, etc.), ce qui nous empêche d'exprimer le dégoût par exemple (voir figure 2.4[6]). En plus, ce langage favorise la confusion dans le cas où deux émotions sont très proches en termes d'expression comme c'est le cas par exemple de la peur et de la surprise.

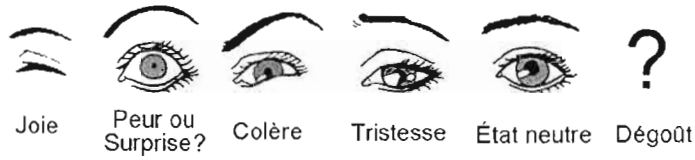

Figure 2.4 Expression des émotions avec un œil.

Par contre, si on modélise le visage en entier, notre expression va être meilleure (voir figure 2.5[7]) ; mais la structure étant plus complexe, l'analyse prendra plus de temps à vérifier puisqu'il faudra analyser chaque élément du visage avant de prendre une décision sur la nature de l'émotion.

---

[6] http://www.jecommunique.com

[7] http://www.iforum.umontreal.ca

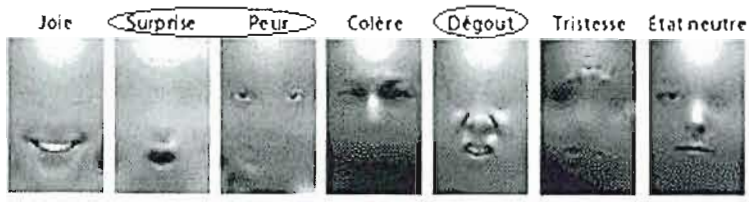

Figure 2.5 Expression des émotions avec un visage.

2.4 Détection d'intrusions dans les réseaux sans-fil

Comme on l'a mentionné au début de ce chapitre, il est important de connaître les spécificités du système à surveiller pour savoir quelle méthode d'analyse est la plus appropriée. Dans le cas des réseaux locaux sans-fil 802.11, on a retenu les particularités suivantes :

- Un débit moins élevé que les réseaux filaires (802.11g : 54 Mbps). Ceci a un impact certain sur le choix de la méthode d'analyse puisqu'on a moins de données à analyser.
- Un environnement favorisant la mobilité des systèmes. Ceci implique le besoin de corrélation. En effet, dans un tel environnement, il faut prendre du recul par rapport à l'infrastructure réseau pour arriver à détecter des attaques fragmentées en plusieurs morceaux. Par exemple, on pourrait imaginer une attaque *ChopChop* où chaque devinette serait effectuée sur un point d'accès différent. Ainsi, un détecteur d'intrusions qui n'est pas capable de faire la corrélation entre les événements observés sur les différents points d'accès ne pourra pas détecter ce type d'attaques.

On présente dans ce qui suit les outils Kismet et Snort-Wireless spécialisés dans la détection d'intrusions dans les réseaux sans-fil. On présente aussi les principales caractéristiques d'Orchids, outil qui sera discuté plus en détails dans les chapitres suivants.

2.4.1 Kismet

Kismet est à la fois un renifleur, un outil d'attaque sur le protocole WEP et un détecteur d'intrusions pour les réseaux sans-fil 802.11. Il a été développé en 2001 par Mike Kershaw (alias Dragorn). L'outil est codé en C/C+− et est distribué sous la licence GPL *(GNU General Public License)*. La version étudiée ici est la Kismet-2006-04-R1 sortie le 13 avril 2006 [15].

Cette version définit une dizaine de signatures d'attaques codées en dur dans l'outil. En effet, Kismet ne propose pas de langage de signatures. Ainsi, il faudrait à chaque fois modifier le code source et recompiler l'outil pour lui permettre de détecter une nouvelle attaque.

Tout le système repose sur une grosse fonction nommée ProcessPacket appelée systématiquement dans une boucle while(1). Dans cette fonction, une longue série d'instructions if/else imbriquées permet de filtrer les paquets et donc de détecter les attaques.

2.4.2 Snort

Snort a été créé en 1998 par Martin Rœsch[8]. Distribué sous la licence GPL, cet outil est vite devenu la référence en matière de détection d'intrusions. En 2001, tout en restant actif sur son projet *open source*, Martin a fondé Sourcefire Inc., une entreprise qui vend des services et des produits de sécurité, notamment Sourcefire 3D System qui est basé en grande partie sur Snort.

Le module 802.11[9] a, quant à lui, été développé par Andrew Lockhart en 2003. Il a été codé en langage C et est distribué avec une dizaine de signatures d'attaques spécifiques aux réseaux sans-fil 802.11. En fait Snort-Wireless n'est pas une nouvelle solution de détection d'intrusions, mais simplement une extension de Snort lui permettant de prendre en considération des événements spécifiques à ce type de réseaux. Cette extension concerne aussi le langage de signatures proposé par Snort.

En effet, Snort-Wireless utilise une syntaxe semblable à Snort, la seule différence est qu'au lieu de préciser les adresses IP source et destination et les ports source et destination, on indique les adresses MAC *(Medium Access Control)* source et destination en suivant la notation hexadécimale avec le séparateur «:» (par exemple 00:03:47:17:14:4C). Voici la syntaxe des règles dans Snort-Wireless :

```
<action> wifi <mac> <direction> <mac> (<rule options>)
```

Snort-Wireless offre aussi les mêmes actions de base qu'on trouve avec Snort : `alert`, `log`, `pass`, `activate` et `dynamic`. Par contre, de nouvelles options de règles (`<rule options>`) spécifiques au protocole `wifi` sont définies en plus des options standard de Snort : `wep`, `ssid`, `pwr_mgmt`, `frame_control`, etc.

---

[8] www.snort.org

[9] `www.snort-wireless.org`.

La figure 2.6 illustre le processus de traitement des paquets dans Snort. Les paquets passent en premier par le *Rule Optimizer* qui va sélectionner un ensemble approprié de règles relativement à la nature du flux observé. Ensuite, le *Multi-Rule Inspection Engine* cherche les règles qui concordent avec le paquet analysé et construit une liste avec ces règles. Dans le cas où plus d'une règle concorde avec le paquet, le *Multi-Rule Inspection Engine* sélectionne la meilleure règle en se basant sur des priorités prédéfinies.

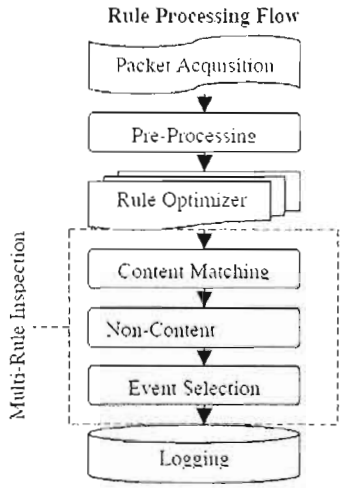

Figure 2.6 Processus de traitement des paquets dans Snort (figure tirée de [21]).

2.4.3 Orchids

Orchids est un détecteur d'intrusions par reconnaissance de scénarios qui offre de nouvelles méthodes de corrélation basées sur une logique temporelle. Il a été développé par Julien Olivain au LSV (Laboratoire Spécification et Vérification) à l'ENS-Cachan (École Normale Supérieure de Cachan) sous la supervision de Jean Goubault-Larrecq depuis décembre 2002. L'outil est capable d'analyser et de corréler plusieurs événements dans le temps et de prendre des contre-mesures.

Nous avons amorcé le développement de l'extension sans-fil pour Orchids à l'automne 2005. On discutera plus à fond des caractéristiques d'Orchids au chapitre 4 et les détails de mise en œuvre du module sans-fil au chapitre 5. Avant cela, on présente dans le chapitre suivant les fondements de la logique utilisée dans Orchids pour effectuer le travail de détection, soit la vérification formelle de modèles.

CHAPITRE III

VÉRIFICATION FORMELLE DE MODÈLES

Plus connue sous son nom anglophone de *model-checking*, la vérification de modèles est une discipline qui a vu son essor dans les années 80 avec les travaux de Clarke et Emerson [8]. La plupart du temps elle est présentée comme un outil qui va permettre d'éviter une catastrophe en évoquant quelques échecs historiques, «comme par exemple la panne du réseau téléphonique aux USA en 1989 ou la destruction du premier exemplaire de la fusée Ariane 5 en 1996» [29].

En effet, la vérification de modèles est généralement présentée en lien avec les domaines du génie logiciel, des télécommunications ou de la microélectronique en tant que technique qui permet de vérifier des propriétés sur un modèle dans le but de détecter une anomalie ou une faiblesse dans un système donné (logiciel, protocole de communication, circuit intégré, etc.). Mais, les récents travaux de Jean Goubault-Larrecq, Muriel Roger et Julien Olivain [23; 26] montrent une nouvelle façon intéressante d'appliquer les techniques de vérification de modèles dans le domaine de la détection d'intrusions, ce qui sort du cadre habituel.

La vérification de modèles est basée sur trois étapes : la modélisation du système réel, la spécification des propriétés à vérifier et finalement la vérification proprement dite. La figure 3.1 montre l'exemple d'une mobylette modélisée par un dessin à la main faisant abstraction de plusieurs éléments du système réel. La propriété «Je peux immobiliser la mobylette à tout moment» est spécifiée dans un langage naturel et la vérification est effectuée par un être humain qui va vérifier si cette propriété est valide dans le modèle.

Dans ce qui suit, on présente la vérification de modèles dans sa forme la plus générale, indépendamment de tout domaine d'application. On présentera dans la première section la phase de modélisation en s'intéressant particulièrement à la modélisation des systèmes réactifs. Dans la deuxième section, on présente la phase de spécification des propriétés à vérifier. On termine le chapitre en présentant des outils usuels pour la vérification automatique de modèles.

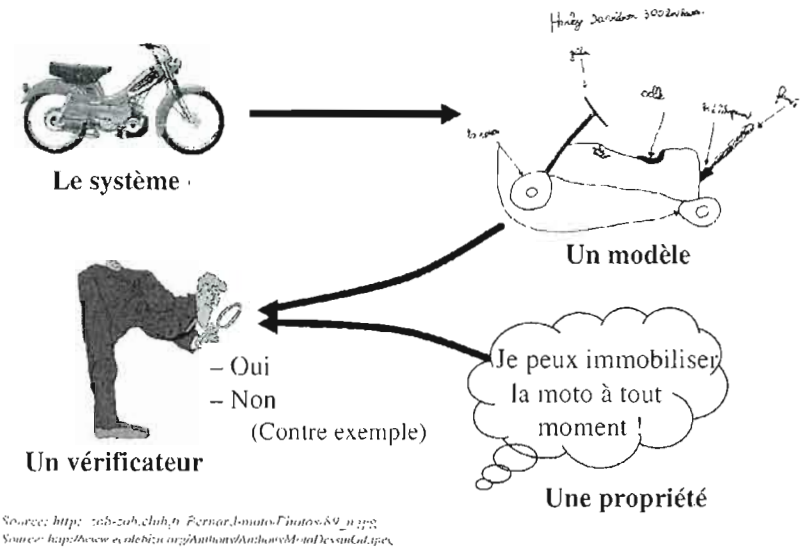

Figure 3.1 Étapes de la vérification de modèles.

3.1 Modélisation

La première étape pour la vérification de modèles consiste à modéliser le système réel (protocole, algorithme, circuit, etc.) à l'aide d'un langage particulier. Étant donné la diversité des systèmes et des objectifs de vérification, il existe de nombreux langages de modélisation : réseaux de Petri, algèbre des processus, Esterel, *statecharts*, automates, etc.

Manna et Pnueli classent les systèmes en deux catégories : réactifs et transformationnels. Comme l'explique Guy Tremblay[1] :

> Un système est dit *réactif* lorsqu'il maintient une interaction constante avec son environnement et lorsque son comportement est dirigé par les événements *(event-driven)*. Les événements sont liés soit à des stimuli internes ou externes, soit à des contraintes liées à l'écoulement du temps. Par opposition, un système est dit *fonctionnel* (ou transformationnel) lorsqu'il produit à partir de données d'entrée un ensemble de sorties puis *termine* son exécution. De tels systèmes sont aussi dit *input-output driven* (plutôt que *event-driven*).

[1] Guy Tremblay. Notes de cours pour MGL7160 (Méthodes formelles et semi-formelles). Dép. d'informatique, UQÀM, Hiver 2002.

Dans cette section, on s'intéresse à la modélisation des systèmes réactifs. Une des méthodes de modélisation de ce type de systèmes est la modélisation par automates finis. Le modèle est, dans ce cas, une description simplifiée du comportement du système à vérifier. Il est donc possible de modéliser le même système de plusieurs façons selon les besoins et selon la connaissance de la réalité physique du système.

Sommairement, un automate est schématisé par un graphe orienté. Les sommets du graphe représentent les états du système modélisé et les arcs représentent les transitions possibles entre ces états. L'état initial est identifié à l'aide d'une flèche incidente. Par exemple, une mobylette peut être représentée par l'automate de la figure 3.2 où chaque sommet affiche l'état du moteur et des roues : au repos ou en marche, et les transitions indiquent les actions possibles sur la mobylette : pédaler, freiner, démarrer, arrêter, etc.

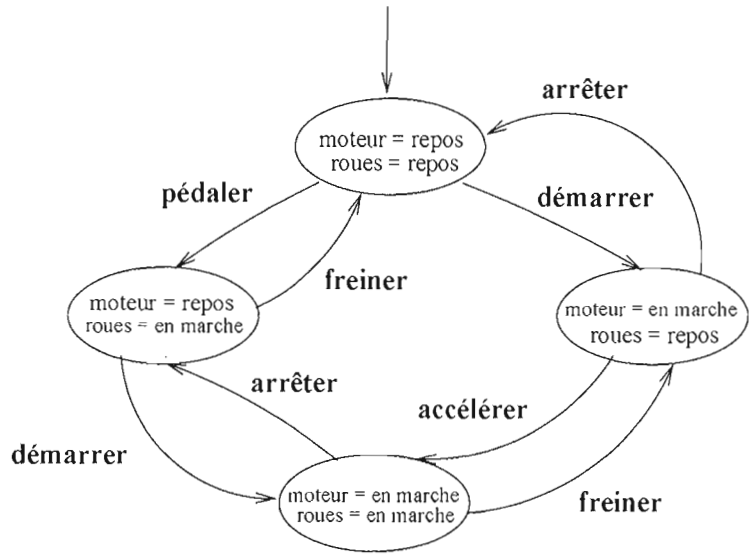

Figure 3.2 Un modèle simple d'une mobylette.

Selon que l'on porte plus d'intérêt aux états qu'aux transitions ou l'inverse, on parlera respectivement de structures de Kripke ou de systèmes de transitions. En effet, les structures de Kripke n'attachent aucune étiquette aux transitions, alors que pour les systèmes de transitions ce sont les états qui sont anonymes. Cependant, il est facile de passer d'un modèle à un autre, comme il est aussi possible de formuler des modèles hybrides tels que celui de l'exemple de la figure 3.2.

On appelle un enchaînement, finie ou infinie, d'états et de transitions «une exécution» (on parle aussi d'«un chemin»). Chaque «exécution» décrit une évolution possible du système modélisé. Ainsi $\left(\frac{moteur=repos}{roues=repos}\right) \xrightarrow{pédaler} \left(\frac{moteur=repos}{roues=en\ marche}\right) \xrightarrow{démarrer} \left(\frac{moteur=en\ marche}{roues=en\ marche}\right)$, et $\left(\frac{moteur=en\ marche}{roues=repos}\right) \xrightarrow{accélérer} \left(\frac{moteur=en\ marche}{roues=en\ marche}\right)$ représentent des exécutions possibles du modèle de la mobylette.

L'ensemble de toutes les exécutions possibles du système peut être représenté sous forme d'un arbre (voir figure 3.3). Dans l'exemple de la mobylette, la racine est l'état initial $\left(\frac{moteur=repos}{roues=repos}\right)$. À partir de cet état, on peut atteindre soit $\left(\frac{moteur=repos}{roues=en\ marche}\right)$ ou $\left(\frac{moteur=en\ marche}{roues=repos}\right)$ en exécutant respectivement les actions pédaler ou démarrer. On recommence ensuite avec ces nouveaux nœuds qui ont, chacun, deux fils étiquetés par $\left(\frac{moteur=en\ marche}{roues=en\ marche}\right)$ et $\left(\frac{moteur=repos}{roues=repos}\right)$. «On se retrouve ainsi avec une représentation infinie de l'ensemble des exécutions du système» [29]. Cependant, l'automate lui même est fini puisqu'il possède un nombre fini d'états.

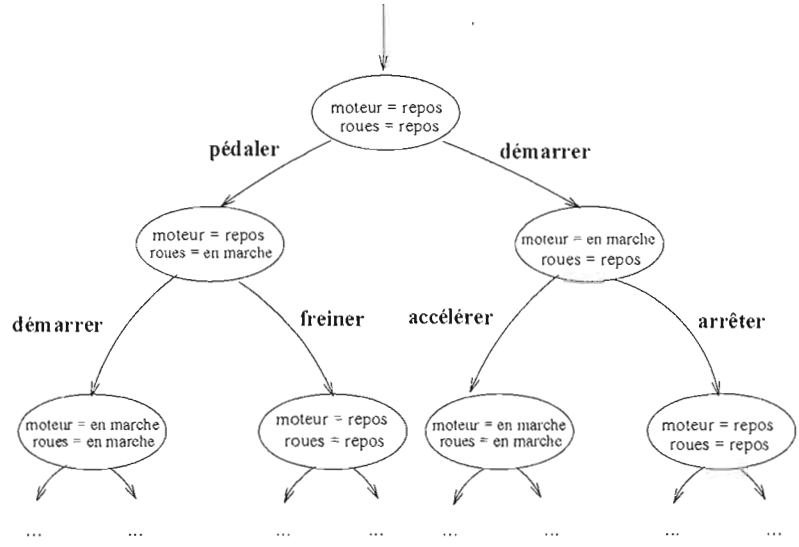

Figure 3.3 Début de l'arbre des exécutions associées au modèle de mobylette de la figure 3.2.

3.2 Spécification des propriétés à vérifier

Avant de commencer la vérification proprement dite, il faut identifier ce qu'on souhaite vérifier. La littérature parle généralement de propriétés comme la sûreté, la vivacité, l'équité, etc. Pour spécifier les propriétés, on utilise un langage particulier (déclaratif) adapté au choix du formalisme de modélisation effectué dans la première étape. En général, on utilisera une logique temporelle.

Les logiques temporelles font partie d'une famille de langages «spécialisés dans les énoncés et raisonnements faisant intervenir la notion d'ordonnancement dans le temps» [29] (le temps est discret dans ce cas). Elles offrent des «opérateurs calqués sur des constructions linguistiques comme les adverbes, les temps de la conjugaison des verbes, etc., de sorte que les énoncés en langue naturelle et leur formalisation en logique temporelle soient assez proches» [29].

Il existe plusieurs logiques temporelles. «LTL *(Linear Temporal Logic)* et CTL *(Computation Tree Logic)* sont deux des logiques temporelles les plus utilisées dans les outils de vérification formelle» [29]. «Ces deux logiques sont interprétées sur des structures de Kripke» [18] et sont donc basées sur les états (par opposition aux logiques temporelles basées sur les actions et interprétées sur des systèmes de transitions).

Bien que l'objet de notre mémoire porte principalement sur la logique LTL sur laquelle est basé Orchids, on présentera dans ce qui suit les deux logiques avec des exemples des propriétés qu'elles permettent d'exprimer.

3.2.1 LTL (Linear Temporal Logic)

Comme on vient de le voir plus haut, une exécution peut être décrite par une suite d'états successifs. Au niveau de chaque état on a des propositions qui peuvent être évaluées à vrai ou à faux. On verra dans les paragraphes suivants comment LTL nous permet d'exprimer des valeurs de vérité combinées dans le temps à l'aide de ses trois principaux opérateurs temporels X, F et G (en plus des opérateurs booléens usuels).

«Tandis que P énonce une proposition vraie dans l'état courant, XP énonce qu'à la prochaine étape de l'évolution du système, l'état suivant (X pour *neXt*) vérifie P.» [29] Par exemple, la formule «$moteur = repos \wedge XXmoteur = repos$» énonce que le moteur de la mobylette est «maintenant» au repos, et qu'il sera encore au repos après deux transitions. En vérifiant l'arbre d'exécution de la mobylette à la figure 3.3, on peut remarquer que la formule est vraie au niveau de l'état initial pour les deux exécutions suivantes : $\left(\frac{moteur=repos}{roues=repos}\right) \xrightarrow{pédaler} \left(\frac{moteur=repos}{roues=en\ marche}\right) \xrightarrow{freiner} \left(\frac{moteur=repos}{roues=repos}\right)$, et $\left(\frac{moteur=repos}{roues=repos}\right) \xrightarrow{démarrer} \left(\frac{moteur=en\ marche}{roues=repos}\right) \xrightarrow{arrêter} \left(\frac{moteur=repos}{roues=repos}\right)$.

Pour exprimer un futur qu'on ne connaît pas précisément, on utilise l'opérateur F. Ainsi, FP énonce que durant les prochaines étapes d'évolution du système, un état futur (F pour

Future) vérifie P (sans préciser quel état exactement). On écrira par exemple *«F moteur = repos»* pour dire qu'à partir de l'état courant (maintenant), on sera ultérieurement (plus tard) dans un état avec un moteur au repos.

Si on veut préciser que cet énoncé reste toujours vrai, c'est-à-dire qu'à tout moment (à partir de n'importe quel état et non seulement maintenant) on peut toujours atteindre plus tard un état avec un moteur au repos, on écrira *«G(F moteur = repos)»*.

Comme son nom l'indique, le temps est linéaire dans cette logique. En d'autres termes, LTL ne s'intéresse qu'à une exécution particulière à un moment donné et donc le futur est unique (on parle aussi d'un futur déterministe, connu à l'avance). Au fait, la représentation en arbre d'exécution n'est pas très fidèle à l'idée de linéarité, il faudrait plutôt imaginer une représentation en «pluie d'états» comme dans la figure 3.4.

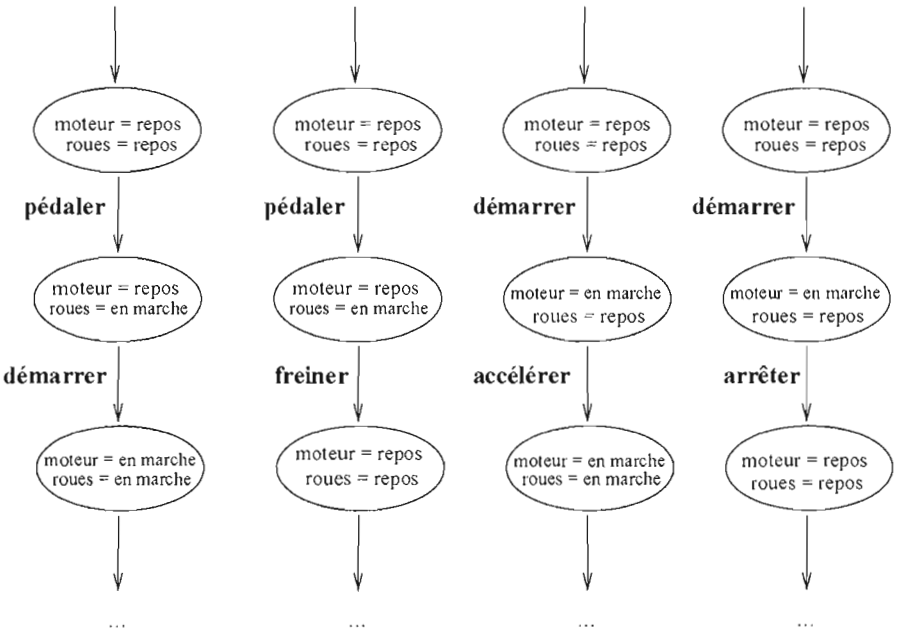

Figure 3.4 Représentation en «pluie d'états» de l'automate de la figure 3.2.

3.2.2 CTL (Computation Tree Logic)

CTL est une logique temporelle qui permet d'exprimer le côté arborescent du comportement. Plusieurs futurs sont possibles à partir d'un moment donné, c'est le futur habituel non-déterministe. La représentation en arbre d'exécution est donc plus appropriée dans ce cas.

Avant de présenter les opérateurs spécifiques à la logique CTL, on va introduire les deux quantificateurs A et E qu'on trouve dans la logique CTL*[2]. Les quantificateurs spécialisés A et E permettent de quantifier sur l'ensemble des exécutions (on les appelles aussi quantificateurs de chemins). Par exemple, si on considère une formule P exprimant une propriété de l'état, la formule AP énoncerait que toutes les exécutions partant de l'état courant satisfont P, tandis que EP énoncerait qu'à partir de l'état courant, il existe une exécution satisfaisant P.

En fait, on peut voir l'opérateur A comme l'équivalent de l'opérateur G de LTL appliqué sur l'ensemble des chemins constituant l'arbre d'exécution au lieu de l'ensemble des états d'un chemin donné. De la même façon, l'opérateur E peut être considéré comme équivalent à l'opérateur F de LTL appliqué sur l'ensemble des chemins. «Plus généralement, A et E quantifient sur les chemins, F et G quantifient sur les états le long d'un chemin donné.» [29]

Les opérateurs CTL sont obtenus en précédant chaque opérateur LTL par un quantificateur de chemin. Par exemple, EFP exprime qu'il est *possible* (en suivant l'une des exécutions) d'avoir P un jour. AFP exprime que l'on aura *forcément* P un jour (quelle que soit l'exécution retenue)» [29]. La figure 3.5 illustre quelques formules CTL courantes et montre la différence avec la logique LTL.

3.3 Vérification

L'étape suivant la modélisation du système et la spécification des propriétés est l'étape de vérification. Plusieurs outils permettent d'effectuer la vérification de modèles de façon automatique et complète (d'où l'intérêt). Chaque outil met en œuvre une logique spécifique en se basant sur un langage de modélisation et un langage de spécification particuliers.

Par exemple, l'outil SMV *(Symbolic Model Verifier)* [7] permet d'interpréter la logique CTL sur des structures de Kripke décrites par des variables d'états et des relations de transition. Il utilise les BDD *(Binary Decision Diagram)* pour représenter les relations de transitions (modèle symbolique).

Un autre outil, populaire dans la vérification automatique de modèles, est Spin [13]. Il permet d'effectuer la modélisation à l'aide de Promela (*PROcess MEta LAnguage*) — un langage de description de processus avec canaux de communication — et la spécification des propriétés à l'aide de LTL.

---

[2] CTL* est une logique qui englobe LTL et CTL.

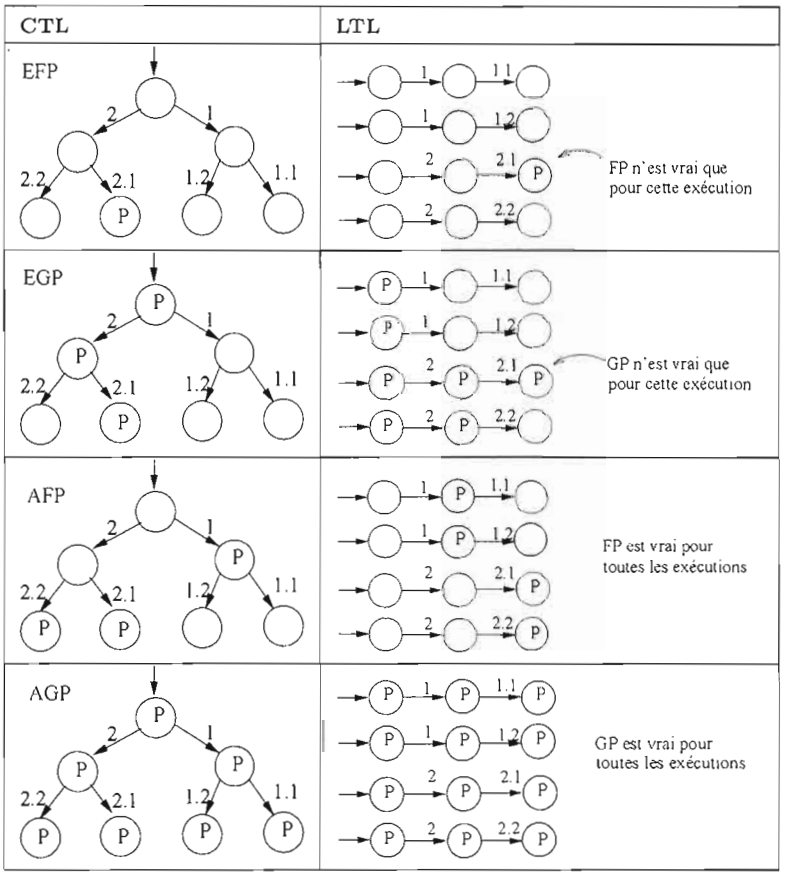

Figure 3.5 Correspondance entre des formules CTL et LTL. (inspiré de la figure 2.2 de [29])

Comme on va le voir au chapitre suivant, par rapport à ces outils de vérification de modèles, Orchids peut être considéré comme un vérificateur de traces. En effet, Orchids définit seulement un langage de spécification de propriétés et au lieu de modéliser le système à vérifier, il analyse plutôt une trace d'exécution effective de façon linéaire.

CHAPITRE IV

ORCHIDS

Le projet Orchids a démarré en décembre 2002 sous la supervision de Jean Goubault-Larrecq dans le cadre du RNTL-DICO (Réseau National des Technologies Logicielles - Détection d'Intrusion Coopérative). L'outil Orchids a été développé par Julien Olivain au LSV (Laboratoire Spécification et Vérification) à l'ENS-Cachan (École Normale Supérieure de Cachan).

Orchids est un détecteur d'intrusions par reconnaissance de scénarios qui offre de nouvelles méthodes de corrélation basées sur une logique temporelle. Il est capable d'analyser, de corréler plusieurs événements dans le temps et de prendre des contre-mesures. Le processus d'analyse est basé sur l'algorithme décrit dans «*Log auditing through model checking*» [26], auquel ont été apportées quelques optimisations et ajoutées de nouvelles options comme les *cuts*, l'opérateur *without*, etc. L'outil est distribué sous la licence de logiciel libre CeCILL[1] (version 2) soumise au droit français.

Pour faire le lien avec le chapitre précédent, on montre dans la section suivante comment la vérification de modèles s'applique dans le cadre de la détection d'intrusions avec Orchids. Ensuite, on présente dans la deuxième section l'architecture générale du logiciel. Dans la troisième section, on expose la logique utilisée pour la spécification des intrusions, et pour finir on illustre, dans la quatrième section, le processus de vérification.

---

[1] Acronyme pour CEa Cnrs Inria Logiciel Libre. Le Commissariat à l'Énergie Atomique, le Centre National de la Recherche Scientifique et l'Institut National de Recherche en Informatique et en Automatique ont élaboré CeCILL en conformité avec le droit français en reprenant les principes de la GNU GPL. On peut trouver le texte officiel de la licence sur le site Internet à l'adresse `http://www.cecill.info`.

4.1 La vérification de modèles dans Orchids

On a vu dans le chapitre précédent que pour effectuer des opérations de *model-checking* il faut commencer par modéliser le système réel à l'aide d'un langage abstrait, ensuite on spécifie les propriétés qu'on veut vérifier à l'aide d'un langage déclaratif, et finalement on fait appel à un vérificateur pour vérifier si ces propriétés sont valides dans le modèle défini.

Dans Orchids, le système qu'on voudrait modéliser est constitué par toute entité dynamique dans le système informatique : les fichiers de journalisation de tout genre, les flux réseaux, les systèmes de gestion de bases de données, etc. (voir figure 4.1). Il est clair que la modélisation d'un tel système serait une tâche complexe et fastidieuse. Par contre, on a immédiatement accès à une trace d'exécution du système : c'est la séquence des événements produits et observés dans le temps (on suppose que deux événements distincts ne peuvent pas se produire exactement au même moment, l'ordre est donc total).

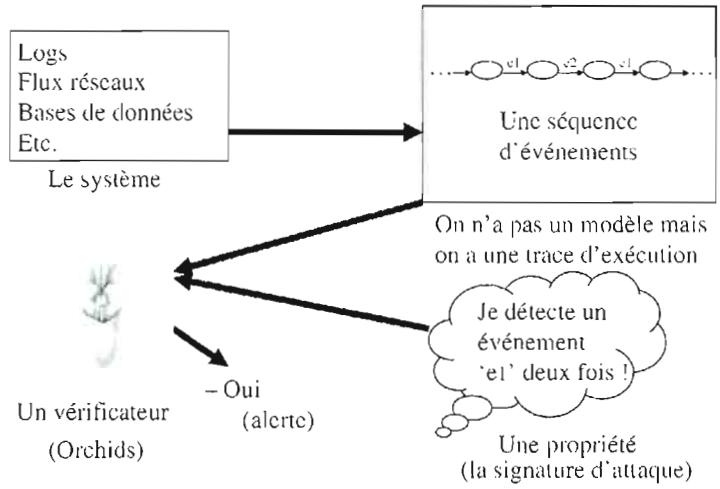

Figure 4.1 Vérification de modèles dans Orchids.

Les propriétés à vérifier, qui ne sont rien d'autre que les signatures d'attaques qu'on voudrait détecter, sont exprimées dans un langage d'automates proche de LTL (vu qu'on travaille sur une seule trace d'exécution). Plus précisément, les personnes qui ont conçu Orchids ont choisi d'utiliser uniquement l'opérateur F de LTL et ont par conséquent éliminé les opérateurs G et X. On s'attardera plus sur la syntaxe et la sémantique du langage de spécification des intrusions dans la section 4.3.2.

Finalement, Orchids vérifie les propriétés définies sur la trace d'exécution du système (la séquence d'événements produite par le système), et lance une alerte dans le cas où il trouve une concordance.

4.2 Architecture générale d'Orchids

Notre objectif ici est de montrer les différentes briques qui constituent Orchids sans entrer dans les détails de leur fonctionnement. Ceci nous permettra d'exposer le contexte de notre propre contribution. On reviendra plus tard pour détailler plus en profondeur certains aspects.

L'architecture globale de la plate-forme Orchids est composée de quatre parties principales (voir figure 4.2) :

- un ensemble de règles qui décrivent les comportements suspects et inhabituels qu'on voudrait détecter ;
- un compilateur de règles qui traduit les définitions des règles en une représentation interne d'automates (l'ensemble de règles compilées constitue alors la base de connaissance de l'outil) ;
- une machine virtuelle pour simuler les automates ;
- un ensemble de modules qui décodent le flux d'informations provenant de plusieurs sources.

Chaque signature d'attaque se trouve, généralement, dans un fichier séparé portant le nom de la règle qui la définit. La syntaxe du langage de spécification des règles est illustrée à la section suivante. Pour l'instant il suffit de comprendre qu'un compilateur traduit chaque règle en un automate pour constituer ainsi la base des signatures d'attaques à détecter.

À partir de cette base et des événements injectés dans le noyau d'Orchids, ce dernier va simuler les automates et tenir à jour les chemins actifs et les états atteints à travers une file ordonnée de *threads*[2]. Ce fonctionnement est explicité dans la section 4.4 «Processus d'analyse».

Les événements injectés proviennent des modules de dissection des données. On a un module spécifique pour chaque type de journal. Ces modules servent principalement à découper les flux d'informations en plusieurs champs. Ces champs peuvent être ensuite référencés dans les définitions des règles.

---

[2]Le mot *thread* ayant un sens particulier dans Orchids, c'est donc ce terme que nous utiliserons.

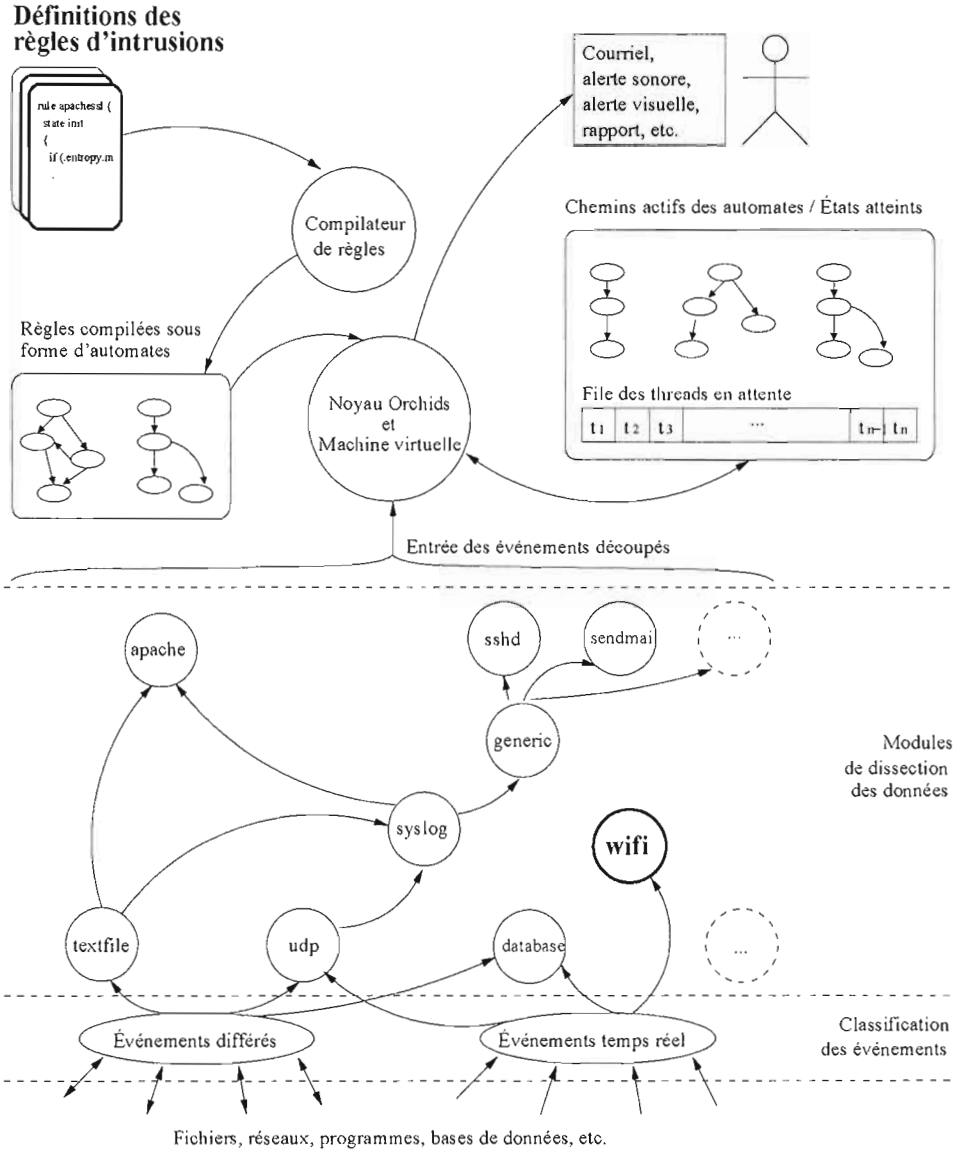

Figure 4.2 Architecture globale d'Orchids (inspirée de [22]).

Notre contribution se situe au niveau des modules de dissection des données et des définitions des règles d'intrusions. En effet, pour pouvoir écrire des signatures d'attaques propres aux réseaux locaux sans-fil 802.11, il faut étendre l'outil de détection d'intrusions pour l'amener à prendre en charge le format de journalisation correspondant. C'est le module **wifi** dont la mise en œuvre est présentée dans le chapitre 5 «Mise en œuvre du module 802.11 et des signatures d'attaques» qui assure cette prise en charge.

4.3 Logique pour la spécification des scénarios d'attaque

On s'intéresse dans cette section au pouvoir expressif du langage de spécification des intrusions offert par Orchids. De ce fait, les exemples présentés ici ne décrivent pas forcément de vraies attaques[3] mais ils permettent en revanche d'illustrer les différentes possibilités d'expression offertes par le langage.

4.3.1 Modélisation de la trace d'exécution

La figure 4.3 met en évidence le rôle joué par les modules — syslog dans ce cas. Comme on peut le constater, le module lit les données dans un fichier texte (il est aussi possible de récupérer les informations dans toute autre forme, notamment dans des paquets réseaux) et les découpe par la suite en différents champs : time, host, program, pid, et message.

Ainsi, il est possible de définir des règles en faisant référence à un champ spécifique découpé par un module spécifique à l'aide de la notation .nom_du_module.nom_du_champ. L'exemple montre une définition de règle permettant de détecter une ouverture de session X sur un système quelconque.

On appelle «enregistrement» toute trace élémentaire produite par une action significative sur un système donné. Par exemple, dans la figure 4.3 chaque ligne du fichier en entrée du module syslog est un «enregistrement».

Un «événement» (au sens d'Orchids) est un «enregistrement» découpé par un module en un ensemble fini de champs avec leurs valeurs respectives et injecté dans le noyau d'Orchids. On

---

[3]Orchids a été testé avec succès sur de vraies attaques récentes et sophistiquées comme l'attaque ptrace (BugTraq ID 7112 http://www.securityfocus.com/bid/7112/) et do_brk (BugTraq ID 9138 http://www.securityfocus.com/bid/9138/).

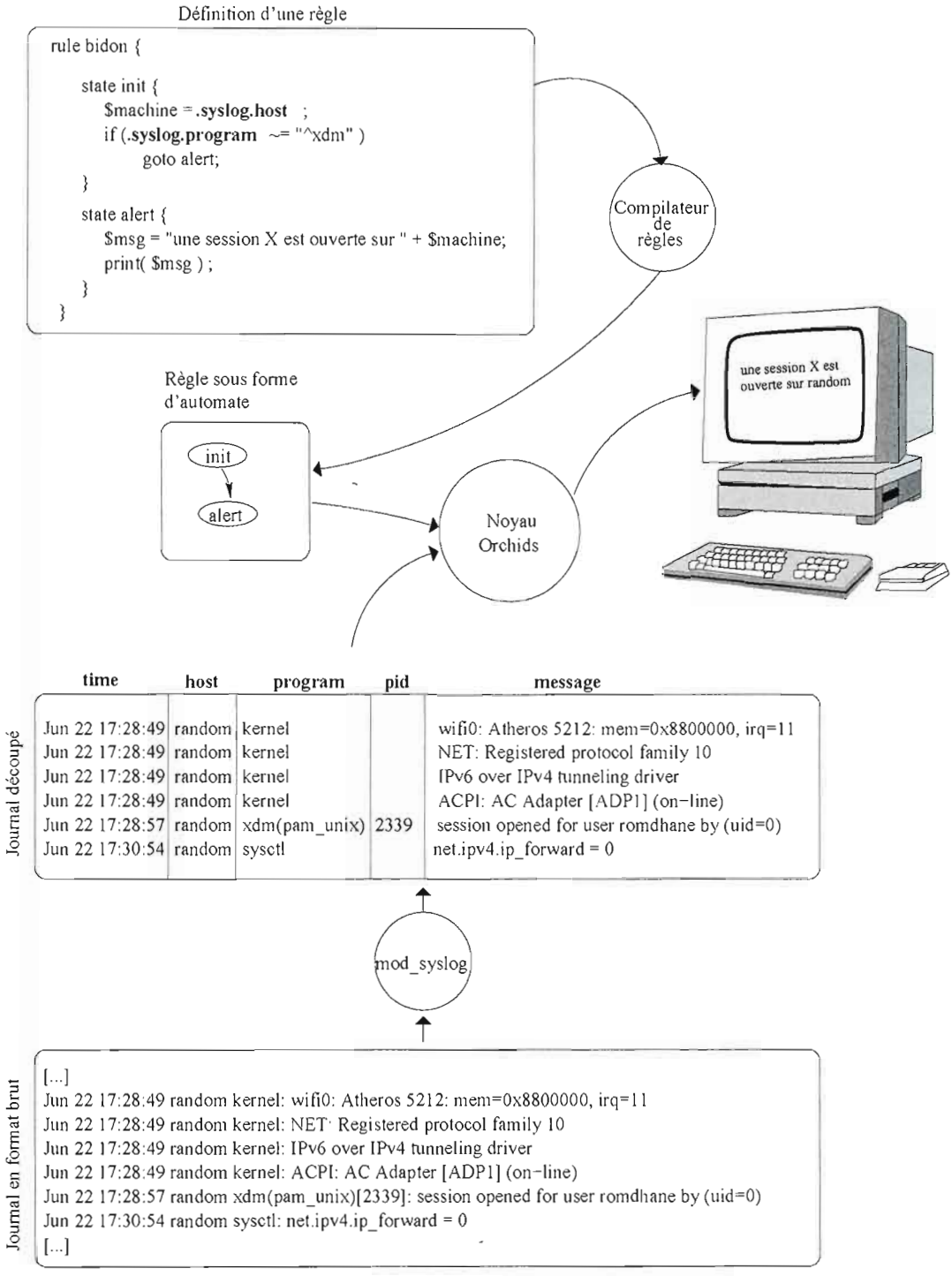

Figure 4.3 Exemple d'un noyau Orchids alimenté par un module syslog et une signature.

parlera donc d'«événements» lorsqu'on veut décrire un aspect dynamique (au cœur du noyau d'Orchids), et d'«enregistrements» pour décrire un aspect statique (plus proche du journal en format brut).

Un «journal» est une séquence, finie ou infinie, d'«enregistrements». Notez que cette définition englobe tout type de journalisation. Ainsi, un paquet réseau peut être considéré comme étant un «enregistrement» et donc tout le flux réseau représente le «journal». Il est aussi intéressant de représenter un «journal», à un niveau abstrait, sous forme de séquence linéaire d'«événements» (voir figure 4.4). C'est sur cette structure qu'Orchids va effectuer la vérification des formules temporelles qui décrivent les signatures d'attaques.

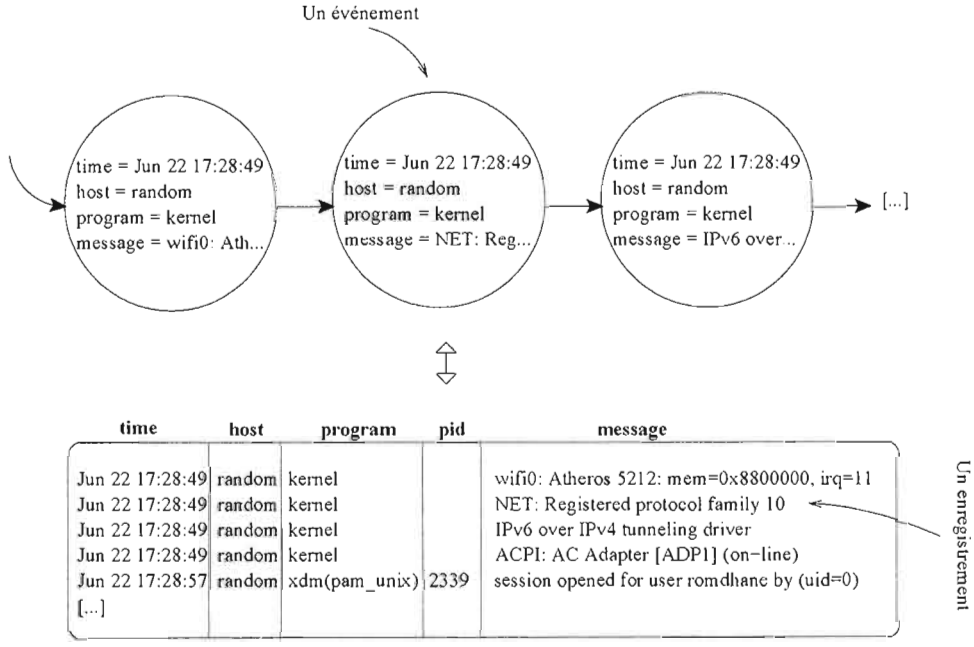

Figure 4.4 Représentation abstraite d'un journal.

4.3.2 Syntaxe et sémantique du langage de spécification

Comme on l'a dit au début de ce chapitre, la spécification des scénarios d'attaques est basée sur une logique temporelle (d'où l'appellation de «formules temporelles») proche du LTL. Les règles sont vues à un niveau abstrait comme des automates finis composés d'un ou plusieurs états, et d'une ou plusieurs transitions à partir de chaque état.

La figure 4.5 représente par exemple l'automate relatif à la signature d'une attaque de capture de mot de passe sur un réseau *(password sniff)*. Chaque règle doit avoir au moins un état initial nommé init, et un ou plusieurs états finaux. À partir de l'état init, et selon les événements reçus (lus dans le journal), on peut passer à travers les états suivants un à un jusqu'à ce qu'on atteigne un état final.

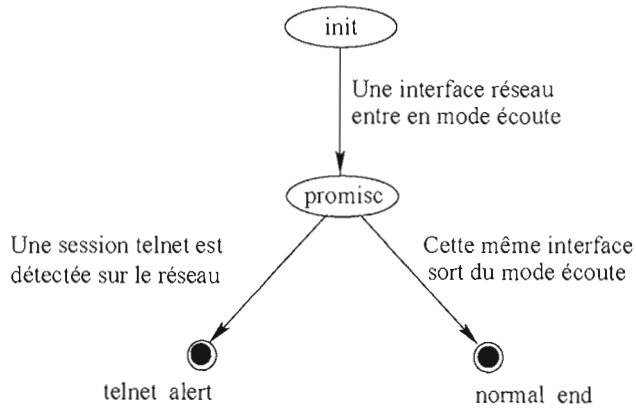

Figure 4.5 Automate de la signature d'attaque password_sniff.

Pour notre exemple, l'événement qui déclenche le passage à l'état promisc est la détection d'une interface réseau qui vient d'entrer en mode écoute (*promiscuous mode*). À partir de cet état, si on détecte que l'interface réseau a quitté le mode écoute, on doit passer à l'état normal_end et donc aucune alerte n'est déclenchée. Par contre, si on détecte le début d'une session Telnet on doit passer à l'état telnet_alert pour aviser l'administrateur système de cette situation (le mot de passe est envoyé en clair dans le protocole Telnet).

Plus concrètement, le fichier signature de l'attaque en question est présenté à la figure 4.6. Le mot clé `rule` permet de définir le nom qu'on veut donner à la règle. Une accolade ouvrante après le nom marque le début du code de la règle. L'accolade fermante en fin du fichier marque la fin du code de la règle. Le code de la règle est composé par plusieurs blocs délimités par une accolade ouvrante et une accolade fermante et débutant par le mot clé `state` suivi du nom de l'état défini. On retrouve ici les états observés dans l'automate illustré à la figure 4.5 : `init`, `promisc`, `telnet_alert`, et `normal_end`.

Pour cette signature, l'état init définit une seule transition conditionnelle «`if` (.promisc.action == "enter") `goto` promisc» de la forme «`if` (condition) `goto` etat_destination». Ceci veut dire qu'à la réception d'un événement qui satisfait la condition de transition, on passe à l'état

```
rule password_sniff {

  state init {
    if ( .promisc.action == "enter" )
      goto promisc;
  }

  state promisc {
    $interface = .promisc.interface;

    if ( .promisc.action == "left" &&
         .promisc.interface == $interface )
      goto normal_end;

    if ( .xinetd.action == "START" &&
         .xinetd.service == "telnet" )
      goto telnet_alert;
  }

  state telnet_alert {
    print("warning, telnet while someone is sniffing");
  }

  state normal_end {
  }

} // end rule
```

Figure 4.6 Fichier signature de l'attaque password_sniff.

etat_destination. Donc, si l'événement en cours provient du module promisc et que la valeur du champ action de ce module correspond à la chaîne de caractères enter, alors on passe à l'état promisc. Formellement, une transition est définie par la syntaxe présentée à la figure 4.7.

```
transition ::=
  if ( condexpr ) goto STATE_NAME ;  /* transition conditionnelle */
| goto STATE_NAME ;                  /* transition inconditionnelle */

condexpr ::=
  NUMBER                             /* entier */
| STRING                             /* chaîne de caractères */
| VARIABLE                           /* $nom_variable */
| FIELD                              /* .nom_module.nom_champ */
| condexpr + condexpr                /* addition */
| condexpr - condexpr                /* soustraction */
| condexpr * condexpr                /* multiplication */
| condexpr / condexpr                /* division */
| condexpr % condexpr                /* modulo */
| condexpr && condexpr               /* et logique */
| condexpr || condexpr               /* ou logique */
| condexpr == condexpr               /* égalité */
| condexpr != condexpr               /* non égalité */
| condexpr > condexpr                /* supérieur */
| condexpr < condexpr                /* inférieur */
| condexpr >= condexpr               /* supérieur ou égal */
| condexpr <= condexpr               /* inférieur ou égal */
```

Figure 4.7 Syntaxe pour la description des transitions.

Au niveau de l'état promisc, on a une action et deux transitions. L'action «$interface = .promisc.interface» est de la forme «$nom_variable = .nom_module.nom_champ» (les noms de variables commencent toujours par le symbole $). Cette action ne fait que récupérer la valeur du champ .promisc.interface et la met dans la variable $interface.

Ce n'est pas la seule forme d'action possible. En effet, une deuxième forme «nom_fonction(arguments)» est présentée au niveau de l'état telnet_alert. Orchids propose une multitude de fonctions prédéfinies qui répondent à différents besoins spécifiques. Voici une liste non-exhaustive de ces fonctions :

- print : affiche, sur la sortie standard, le message qui lui est passé en paramètre ;

- print_event : affiche, sur la sortie standard, l'événement qui a provoqué la transition vers l'état courant ;
- system : exécute une commande système ;
- str_from_int : convertit un entier en une chaîne de caractères ;
- int_from_str : convertit une chaîne de caractères en un entier ;
- sendmail : envoie un courrier électronique.

Malgré le fait que le langage offre plusieurs formes d'actions, les deux formes illustrées dans notre exemple sont les deux formes les plus utilisées. Formellement, une action est définie selon la syntaxe présentée à la figure 4.8.

```
action ::=
  expr ;

expr ::=
  simple-expr                 /* expression atomique */
| FUNCTION ( params )         /* fonction prédéfinie */
| VARIABLE = expr             /* affectation à une variable */
| expr + expr                 /* addition */
| expr - expr                 /* soustraction */
| expr * expr                 /* multiplication */
| expr / expr                 /* division */
| expr % expr                 /* modulo */

simple-expr ::=
  STRING                      /* type chaîne de caractères */
| NUMBER                      /* type entier */
| VARIABLE                    /* une variable commence toujours par $ */
| FIELD                       /* un champ est défini sous la
                                 forme .nom_module.nom_champ */

params ::=
  /* sans paramètres */
| params, simple-expr         /* un ou plusieurs paramètres */
| simple-expr
```

Figure 4.8 Syntaxe pour la description des actions.

En arrivant à un état terminal (un état sans aucune transition comme telnet_alert ou normal_end), on dira que la règle a été observée dans le journal. En d'autres termes, l'une des séquences d'événements décrites par la règle a été retrouvée dans la séquence des événements injectés dans le noyau Orchids.

4.4 Processus d'analyse

Comme on l'a cité au début de ce chapitre, le noyau d'Orchids est basé sur l'algorithme décrit dans «*Log Auditing through Model Checking*» [26]. Des améliorations ont été ensuite apportées à cet algorithme. On peut trouver la forme rigoureuse et complète de l'algorithme avec sa collection d'optimisations dans «Un algorithme pour l'analyse de logs» [10]. D'autres dispositifs tels que les *cuts* et l'opérateur *without* sont décrits dans «Langages de détection d'attaques par signatures» [11].

On n'a pas l'intention d'illustrer le fonctionnement détaillé de l'algorithme de vérification ici. L'objectif est simplement de présenter l'approche sur laquelle est basé le processus d'analyse, sans entrer dans les détails. Pour cela, on va procéder à une série d'abstractions sur le langage de spécification des attaques pour simplifier l'explication.

La figure 4.9 montre une première abstraction sur un automate consistant à ignorer les actions effectuées au niveau de chaque état et à ne considérer que les transitions (les événements), et une deuxième abstraction consistant à passer d'un système de transitions à une structure de Kripke. On peut maintenant écrire l'automate ainsi simplifié à l'aide de l'expression régulière $E_1[E_2{*}]E_3$. On exprime ici notre volonté de détecter un premier événement E_1 suivi par zéro ou plusieurs événements E_2, suivis par un dernier événement E_3.

La figure 4.10 schématise la simulation pas à pas de l'automate $E_1[E_2{*}]E_3$ sur la trace d'un fichier journal constitué successivement par les événements $E_{1.1}$, $E_{2.2}$, $E_{2.3}$, $E_{1.4}$ et $E_{3.5}$. Le deuxième chiffre en indice sur les étiquettes attribuées aux événements indique l'ordre dans lequel un événement est observé dans le journal.

À la réception du premier événement ($E_{1.1}$), Orchids crée une nouvelle instance de règle. Sachant qu'à partir d'un état E_1 on peut atteindre E_2 ou E_3, Orchids crée un premier *thread* bloqué sur l'événement qui valide la transition vers un état E_2 et un deuxième *thread* bloqué sur l'événement qui valide la transition vers un état E_3.

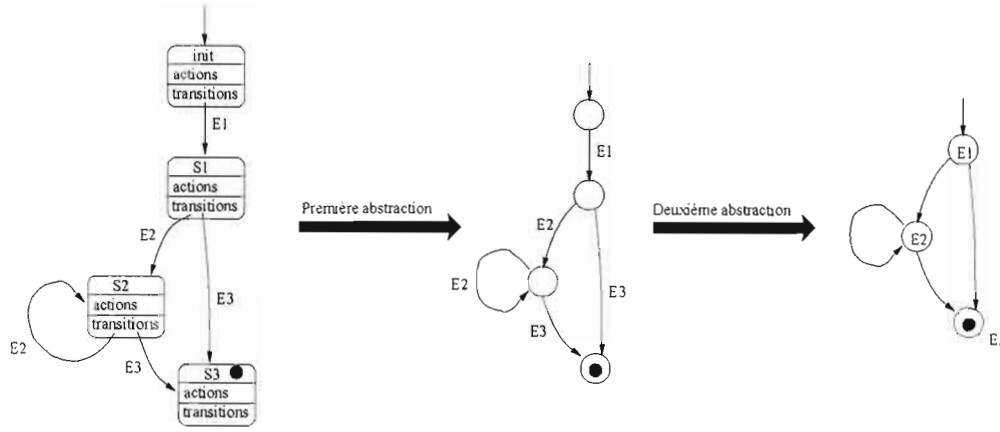

Figure 4.9 Abstractions sur un automate décrit dans le langage de signature d'Orchids.

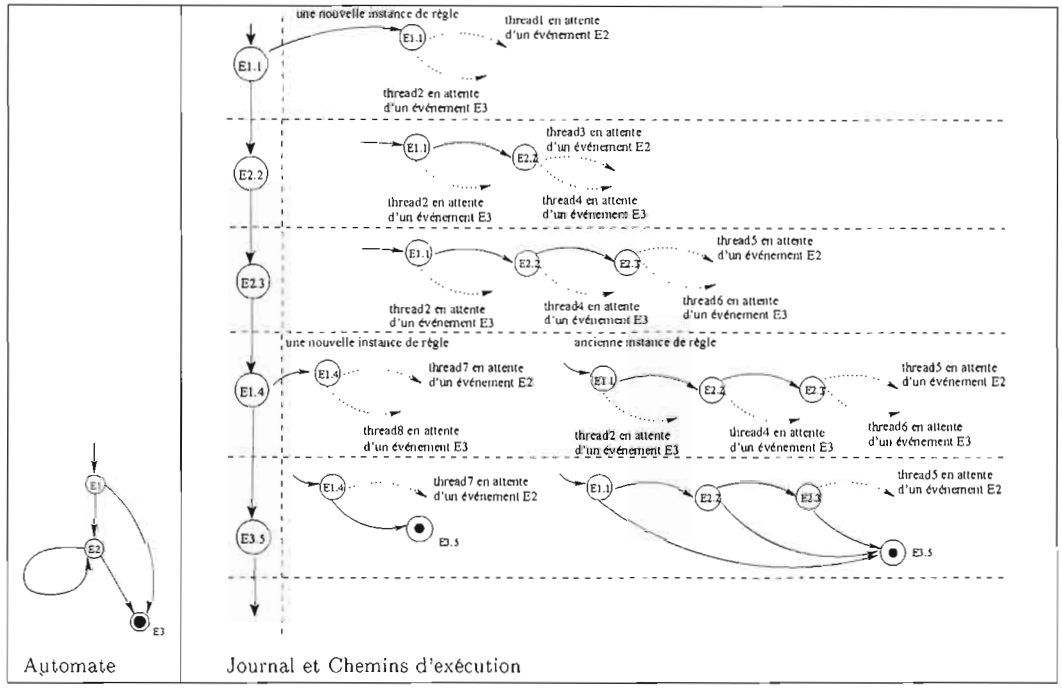

Figure 4.10 Simulation de l'automate $E_1[E_2*]E_3$ sur un journal d'événements.

À la réception de l'événement $E_{2.2}$, le premier *thread* est débloqué ; et puisqu'on peut atteindre les états E_2 ou E_3 à partir de E_2, Orchids crée deux nouveaux *thread* : thread3 en attente d'un événement E_2 et thread4 en attente d'un événement E_3. Deux autres *threads* (thread5 et thread6) sont aussi créés suite à la réception de l'événement $E_{2.3}$ qui débloque par la même occasion le *thread* thread3.

Arrivé au quatrième événement ($E_{1.4}$), une nouvelle instance de règle est créée avec deux nouveaux *threads* : thread7 et thread8.

C'est à la réception de l'événement $E_{3.5}$ (un état terminal de l'automate) que les deux instances se terminent. La deuxième instance de règle se termine avec un seul chemin : $[E_{1.4}E_{3.5}]$. Quant à la première instance, elle se termine avec trois chemins : $[E_{1.1}E_{3.5}]$, $[E_{1.1}E_{2.2}E_{3.5}]$ et $[E_{1.1}E_{2.2}E_{2.3}E_{3.5}]$. Seul le chemin «le plus court», soit $[E_{1.1}E_{2.2}E_{2.3}E_{3.5}]$, est retenu.

Comme on peut le remarquer, le chemin retenu comme étant «le plus court» n'est pas le chemin avec le moins de transitions. Au contraire, c'est bien celui avec le plus grand nombre de transitions. Alors pourquoi est-ce qu'on l'appelle «le chemin le plus court» ?

En fait, le terme «le plus court» veut dire deux choses : «le plus rapide», et celui qui porte «le plus d'informations».

Pour mieux distinguer ces deux notions, prenons un autre exemple plus simple illustré à la figure 4.11. Comme on peut le voir, le chemin le «plus rapide» est celui qui, en partant d'un même état initial, atteint un état final avant un autre chemin (par rapport à l'ordre chronologique du journal).

Revenons maintenant à l'exemple de la figure 4.10. Dans la première instance de règle, tous les chemins sont aussi rapides. En effet, en partant du même état initial $E_{1.1}$, tous les chemins atteignent le même état final $E_{3.5}$ en même temps (à savoir, à l'étape 5). On n'a donc pas de chemin plus rapide qu'un autre. Dans ce cas, le chemin qu'on considère «le plus court» est celui qui contient aussi «le plus d'informations». En d'autres termes, c'est le chemin avec le plus grand nombre d'états (chaque état fournissant un ensemble d'informations).

Ainsi, une seule alerte est lancée par Orchids au lieu d'une alerte pour chaque chemin. Et c'est l'alerte correspondant au «chemin le plus court».

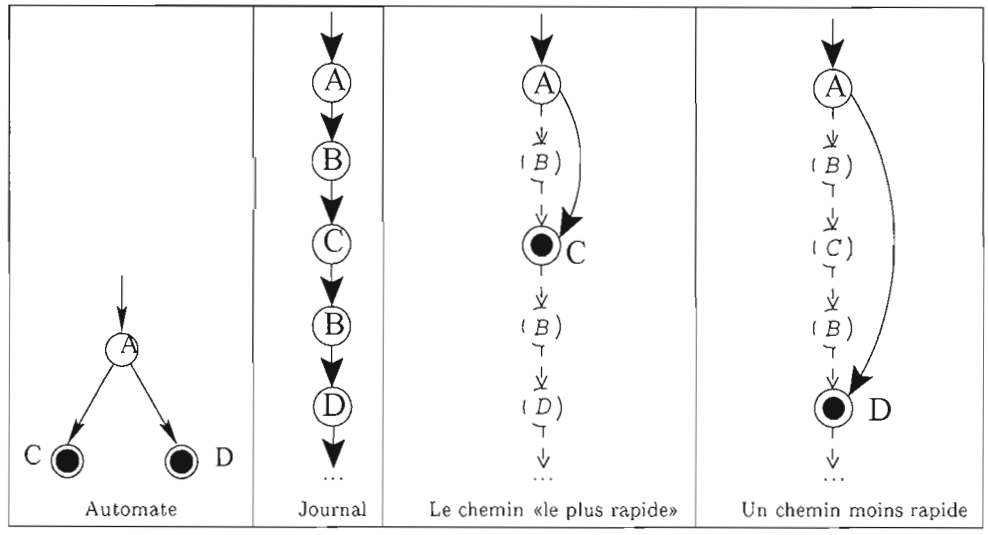

Figure 4.11 Simulation de l'automate A[C|D].

CHAPITRE V

MISE EN ŒUVRE DU MODULE 802.11 ET DES SIGNATURES D'ATTAQUES

On présente dans la première section les choix faits au niveau de l'architecture de la plateforme de détection d'intrusions qui affectent directement les choix de mise en œuvre du module 802.11 discutés dans la deuxième section. Ensuite, on présente dans la troisième section notre démarche pour identifier les signatures d'attaques.

5.1 Architecture de la plateforme de détection d'intrusions

Comme on l'a présenté au chapitre 2, les détecteurs d'intrusions sont généralement composés de deux parties : un ensemble de sondes (pour la collecte des événements) et un ou plusieurs analyseurs. Pour une meilleure sécurité, on place les sondes sur les réseaux qu'on veut surveiller et les analyseurs dans une zone extrêmement protégée.

En attachant un ensemble de sondes à un analyseur, on doit veiller à ce que le nombre d'événements reçus par ce dernier ne dépasse pas sa capacité de traitement, sinon des événements peuvent être rejetés et on pourrait donc passer à côté d'une attaque sans la remarquer.

Un autre point important est de décider de l'endroit où placer les sondes : soit n'importe où dans la zone couverte par le point d'accès, soit dans le point d'accès lui-même. Bien que le choix de mettre la sonde dans le point d'accès semble le plus approprié car c'est le point central de la zone qu'il couvre, on a choisi de la placer juste à côté, pour les raisons suivantes.

Notre premier argument est que le détecteur d'intrusions n'a pas pour tâche de protéger seulement le point d'accès mais aussi les clients du réseau. Il faudrait donc une interface, avec une meilleure sensibilité, qui puisse couvrir une zone plus large que celle couverte par le point

d'accès. Ainsi, un intrus potentiel serait visible par le détecteur d'intrusions même s'il agit en dehors de la zone couverte par le point d'accès.

L'autre argument est que la charge d'analyse est assez considérable. Il faudrait donc concevoir des points d'accès plus performants avec des processeurs plus rapides et une plus grande mémoire. Ceci implique non seulement des coûts supplémentaires, mais aussi une gestion plus lourde de l'équipement. Il ne faut pas aussi sous-estimer le nombre de points d'accès déjà en place et qui ont besoin d'un détecteur d'intrusions indépendant.

En se basant sur l'architecture retenue, on pouvait soit exploiter les journaux du pilote de la carte sans-fil, soit travailler directement sur les trames 802.11 en format brut. Notre choix s'est fixé sur la deuxième possibilité. En effet, les journaux des pilotes ne correspondent à aucun standard et dépendent, donc, fortement de la carte et du pilote utilisés. Généralement, ces journaux comportent des messages au format Syslog reportés, avec un code de priorité, à l'aide de la fonction printk. De plus, pour que la détection soit plus rapide, il est préférable d'analyser directement les trames au format brut.

5.2 Mise en œuvre du module 802.11 pour Orchids

La première opération effectuée durant le lancement du processus de base d'Orchids est la lecture du fichier de configuration `orchids.conf` (le chemin vers ce fichier est précisé durant l'installation ou au moment de l'exécution). Ce fichier est constitué de *directives* et est édité par l'administrateur selon ses besoins spécifiques. La figure 5.1 montre un extrait du fichier de configuration `orchids.conf`. On y observe la directive `SetModuleDir` qui permet d'indiquer le chemin vers les modules Orchids et la directive `LoadModule` qui permet de charger un module spécifique en indiquant son nom (dans l'exemple, on charge les modules wifi et syslog). L'ordre dans lequel apparaissent les directives dans le fichier de configuration est important.

```
# Fichier orchids.conf

SetModuleDir /usr/local/lib/orchids
LoadModule wifi
LoadModule syslog
```

Figure 5.1 Extrait du fichier de configuration `orchids.conf`.

En fait, les modules Orchids sont compilés sous forme de bibliothèques partagées *(shared objects)*. On utilise donc la directive `LoadModule nom_du_module` pour charger en mémoire la bibliothèque partagée `mod_nom_du_module.so`. La figure 5.2 présente une structure d'entrée d'un module Orchids. Cette structure identifie principalement les paramètres suivants pour un module donné :

- Version d'Orchids : pour vérifier la compatibilité du module avec la version d'Orchids utilisée.
- Nom du module.
- Dépendances : liste des modules desquels dépend ce module. Si l'un de ces modules n'est pas encore chargé, le module qui en dépend ne peut pas poursuivre son chargement.
- Directives de configuration : liste des directives de configuration pour les paramètres spécifiques au module. Dans le cas du module wifi, une seule directive spécifique a été développée, à savoir la directive `AddDevice`.
- Fonction de pré-configuration : fonction pour l'initialisation du module. C'est à ce niveau que les champs spécifiques à ce module sont déclarés.

On présente dans ce qui suit les étapes de pré-configuration et de configuration d'un module Orchids ainsi que la mise en œuvre de la capture et du découpage des trames dans le module wifi.

5.2.1 Pré-configuration et configuration d'un module Orchids

Durant le chargement d'un module Orchids, la fonction de pré-configuration indiquée au niveau de la structure d'entrée du module (`wifi_preconfig` dans le cas du module wifi) est invoquée afin de déclarer les champs spécifiques à ce module. On a retenu la liste des champs suivants pour le module wifi : le type de trame, le sous-type, le temps de réception, les différentes adresses (destination, source, point d'accès, transmission et réception), la direction de la trame, le numéro de séquence, une valeur booléenne indiquant s'il s'agit d'une trame chiffrée, la taille des données et, finalement, la puissance du signal reçu.

La figure 5.3 illustre, pour chaque champ retenu, le nom qui lui est associé dans le langage des signatures d'Orchids, son type et un commentaire expliquant en quelques mots sa signification.

```
/**
 ** @struct input_module_s
 **   Input Module Structure.
 **   These modules extract useful information, preprocess and
 **   prepare them to be processed by the analysis engine.
 **/
typedef struct input_module_s input_module_t;
struct input_module_s
{
  unsigned long          magic;
  unsigned long          version;
  char                   *name;
  char                   *license;
  char                   **dependencies;
  struct mod_cfg_cmd_s   *cfg_cmds;
  pre_config_t           pre_config;
  post_config_t          post_config;
  post_compil_t          post_compil;
};
```

Figure 5.2 Structure d'entrée d'un module Orchids.

```
static field_t wifi_fields[] = {
  { "wifi.type",        T_STR,      "frame type"                              },
  { "wifi.subtype",     T_STR,      "frame subtype"                           },
  { "wifi.time",        T_TIMEVAL,  "reception time"                          },
  { "wifi.da",          T_STR,      "destination address"                     },
  { "wifi.sa",          T_STR,      "source address"                          },
  { "wifi.bssid",       T_STR,      "basic service set identifier"            },
  { "wifi.ta",          T_STR,      "transmitter address"                     },
  { "wifi.ra",          T_STR,      "receiver address"                        },
  { "wifi.dir",         T_STR,      "direction of the frame"                  },
  { "wifi.seqnum",      T_INT,      "sequence number"                         },
  { "wifi.iswep",       T_INT,      "1 if it is an encrypted frame, 0 else." },
  { "wifi.bodylen",     T_INT,      "body length"                             },
  { "wifi.rssi",        T_INT,      "received signal strength identification"}
};
```

Figure 5.3 Liste des champs retenus pour le module wifi.

Configurer un module Orchids consiste à appliquer les fonctions associées aux directives, spécifiques à ce module, rencontrées dans le fichier de configuration orchids.conf entre les balises <module nom_du_module> et </module>. La figure 5.4 montre, par exemple, la configuration du module wifi à l'aide de la directive `AddDevice ath0` qui permet d'ajouter une interface sans-fil pour l'écoute d'un flux 802.11. La même figure montre aussi un exemple de configuration du module `syslog` à l'aide des directives `AddTextfileSource`, `AddUdpSource`, `AddUnixSocketSource`.

```
# wifi input module
<module wifi>
  AddDevice ath0
</module>

# syslog input module
<module syslog>
  AddTextfileSource /var/log/messages
  AddUdpSource 10514
  AddUdpSource 514
  AddUnixSocketSource /dev/log
</module>
```

Figure 5.4 Exemple de configuration de modules Orchids.

La fonction associée à la directive `AddDevice` effectue principalement deux opérations : ouvrir l'interface sans-fil, passée en paramètre, en mode écoute à l'aide de la bibliothèque `Pcap` *(Packet Capture library)*, et déclarer la fonction effectuant la création d'événements et leur injection dans le noyau Orchids. On verra dans la section suivante comment se fait la création des événements (toujours au sens Orchids) à partir des trames 802.11.

5.2.2 Capture et découpage des trames 802.11

Comme on vient de le dire, la capture des trames est effectuée à l'aide de la bibliothèque `Pcap`. Avant la capture, on commence par créer un tableau de 13 cases (le chiffre 13 correspond au nombre de champs retenus) de type `ovm_var_t` (type générique propre à Orchids) correspondant aux champs enregistrés. Ensuite, dès qu'on capte une trame, on fixe les valeurs des cases du tableau correspondant aux champs qui sont toujours présents indépendamment du type de la trame capturée, comme par exemple les champs `wifi.rssi` et `wifi.time`. Finalement, on

traverse un long `switch` en fixant à chaque fois les cases du tableau correspondant aux champs spécifiques à la trame capturée. En se basant sur le tableau qu'on vient de remplir, on crée une structure d'un événement et on l'injecte dans le noyau d'Orchids.

5.3 Identification des signatures d'attaques

Pour arriver à écrire correctement une signature d'attaque de façon précise, il faut «*bien connaître l'attaque*» ainsi que ses variantes mais aussi «*bien connaître le système*» surveillé. Bien connaître l'attaque et ses variantes permet de ne prendre que l'essentiel (le minimum requis pour identifier l'attaque le plus tôt possible) et surtout d'éviter les fausses négatives. Bien connaître le système permet d'éviter les fausses positives. En effet, si on ignore les détails de fonctionnement du système, on pourrait croire identifier une attaque alors qu'on ne fait que signaler un comportement normal dans une situation particulière du système.

Bien que le nombre d'attaques détectées par un outil de détection d'intrusions soit un facteur important, on a choisi dans ce mémoire de décrire seulement quelques signatures d'attaques fondamentalement différentes. L'objectif est de montrer les principales fonctionnalités offertes par Orchids, ainsi que certaines de ses limites.

On présente dans les sous-sections suivantes notre démarche pour décrire les attaques *deauthentication flooding* et *ChopChop* à l'aide du langage de signatures d'Orchids, étendu grâce au module wifi. Ces deux attaques ne sont pas les seules qu'on a réussi à identifier et à détecter mais ce sont les seules qu'on va présenter ici. Parmis les attaques qu'on a réussi à identifier et à détecter on cite, par exemple, les attaques *ARP Replay* et *Wellenreiter*.

5.3.1 Identification de l'attaque de désauthentification par inondation

Le fonctionnement de cette attaque a été illustré à la section 1.3.2 «Attaques contre la disponibilité». Dans la présente section, on effectue une analyse détaillée et on présente les propriétés essentielles de l'attaque *deauthentication flooding* ainsi que notre démarche pour l'identifier à l'aide du langage de signatures d'Orchids.

5.3.1.1 Première tentative

On a vu que le comportement observé durant une attaque de désauthentification par inondation correspond à un nombre anormalement élevé de trames du type désauthentification im-

pliquant un ou plusieurs clients en particulier. En effet, lorsqu'un client se trouve désauthentifié il va tenter de se réauthentifier à plusieurs reprises. Pour l'empêcher de reprendre sa connexion, l'intrus peut envoyer continuellement (à une fréquence donnée) des trames de désauthentification. Ainsi, il est possible d'atteindre le temps d'arrêt *(timeout)* d'une couche supérieure (TCP par exemple), ce qui causerait la perte de toutes les sessions du client au niveau applicatif. Mais, il est aussi possible d'effectuer cette attaque avec une fréquence moins élevée, ce qui n'empêchera pas le client de continuer à travailler, mais son débit risque d'être fortement affecté.

La question est donc de savoir quelle est la fréquence à partir de laquelle on peut juger que le comportement est suspect. Il est évident que ce n'est pas normal d'observer cinq désauthentifications du même client durant deux secondes, mais est-ce normal d'en observer cinq durant dix secondes ? Ou deux durant cinq secondes ? Ceci dépend beaucoup de l'architecture en place, des applications utilisées, du matériel et des clients. Le choix de la valeur exacte utilisée pour cet attribut devrait donc être laissé à l'administrateur du réseau.

À ce niveau de l'analyse, on vient de dégager une propriété essentielle de l'attaque *deauthentication flooding* qui est sa fréquence. Pour paramétrer cette propriété dans la signature d'attaque, on a défini deux constantes : NB_FRAMES et DELAY. La valeur attribuée à DELAY indique l'intervalle maximal durant lequel on va chercher une instance de l'attaque, et la valeur attribuée à NB_FRAMES indique le nombre *minimal* de trames de désauthentification qu'il faudrait observer durant l'intervalle DELAY avant de lancer une alerte. Autrement dit, si on observe au moins NB_FRAMES trames de type désauthentification dans un intervalle de temps de DELAY secondes, on lance une alerte. La figure 5.5 présente l'automate correspondant à la signature de l'attaque *deauthentication flooding* qu'on serait tenté de dessiner à ce niveau. On verra dans ce qui suit que cette description n'est pas la meilleure.

5.3.1.2 Problème de multiplication d'alertes

Pour mieux visualiser le problème de multiplication d'alertes, on peut simplifier l'automate de la figure 5.5 en ignorant le paramètre DELAY et en fixant NB_FRAMES à 2, ce qui nous ramène à l'automate de la figure 5.6 (avec d=deauth, L=deauth_loop et A=deauth_alert). Cet automate exprime donc notre volonté de lancer une alerte après l'observation de deux trames de type désauthentification.

La figure 5.7 illustre le résultat de la simulation de l'automate de la figure 5.6 (version simplifiée de l'automate de la figure 5.5) sur la séquence d'événements d_1, d_2, d_3, et d_4 corre-

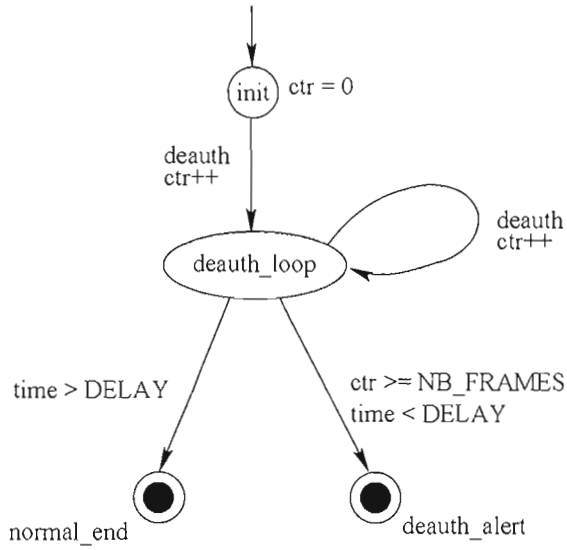

Figure 5.5 Automate correspondant à la signature de l'attaque *deauthentication flooding*.

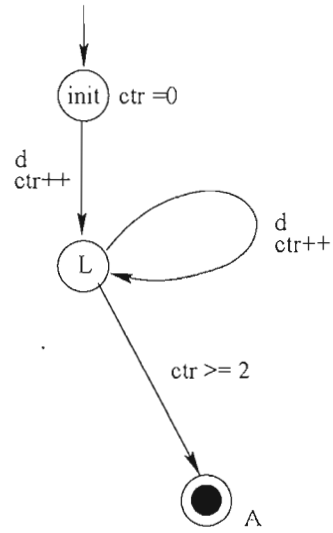

Figure 5.6 Automate simplifié de la signature d'attaque *deauthentication flooding* décrite à la figure 5.5.

spondant à l'observation de quatre trames de type désauthentification provenant de la même source et envoyées vers la même destination. On peut remarquer que trois instances de règles aboutissent à un état d'alerte. La première alerte signale le chemin $[d_1 d_2]$, la deuxième alerte signale le chemin $[d_2 d_3]$ et la troisième alerte signale le chemin $[d_3 d_4]$. Et si une cinquième trame de type désauthentification (d_5) surgit, une nouvelle alerte serait lancée en signalant le chemin $[d_4 d_5]$, et ainsi de suite. Ceci n'est pas le comportement qu'on souhaite observer.

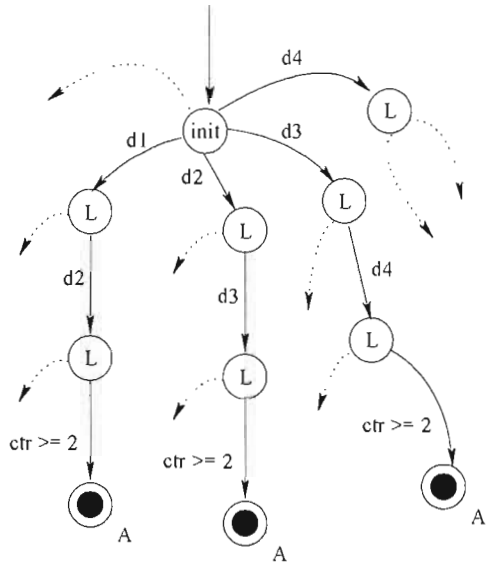

Figure 5.7 Simulation de l'automate de la figure 5.6.

En effet, on voudrait que pour une même attaque on ne retrouve qu'une seule alerte. Cela semble simple à dire mais ce n'est pas aussi facile à formaliser. Avant tout, il faut définir ce qu'on entend par «une même attaque». Dans le cas de la désauthentification par inondation, si l'intrus n'arrête pas d'envoyer des trames de désauthentification, combien d'alertes doit-on trouver dans le rapport d'Orchids ?

On serait tenté de répondre que l'idéal serait une seule alerte dès que l'attaque commence et une notification dès que l'intrus arrête d'envoyer des trames de type désauthentification. Sauf que dans ce cas, un intrus déterminé lancera d'un seul coup une alerte en envoyant NB_FRAMES trames de type désauthentification et attendra DELAY secondes (temps nécessaire pour juger que l'intrus a arrêté l'attaque) avant de recommencer, et ainsi de suite. De cette façon, il réussit à avoir le contrôle sur le remplissage de notre rapport d'alertes. On pourrait aussi envisager de lancer une alerte tous les NB_FRAMES trames ou une alerte tous les DELAY secondes.

Notre choix s'est fixé sur une alerte dès que l'attaque commence, suivie d'un temps d'attente prédéfini `WAITIME` durant lequel aucune nouvelle alerte pour un même couple victime et attaquant ne doit être lancée (le temps d'attente ne doit pas suspendre la détection de nouvelles attaques). Après ce temps, si l'attaque est encore active on lance une nouvelle alerte et on attend de nouveau durant `WAITIME` secondes.

En partant des résultats de la simulation de la figure 5.7, on introduit dans ce qui suit deux constructions offertes par Orchids qui vont nous permettre de mettre en œuvre le choix qu'on vient de se fixer (l'option avec `WAITIME`).

5.3.1.3 Amélioration à l'aide du mot clé `synchronize`

Pour commencer, au lieu d'avoir trois alertes signalant successivement $[d_1 d_2]$, $[d_2 d_3]$ et $[d_3 d_4]$ on aimerait plutôt avoir deux alertes signalant successivement $[d_1 d_2]$ et $[d_3 d_4]$. En d'autres termes, on voudrait mettre en œuvre l'idée consistant à lancer une alerte tous les NB_FRAMES trames. Si on revient à la figure 5.6, on peut voir qu'à la réception d'une nouvelle trame, une nouvelle instance de règle est systématiquement créée ; c'est ce qui fait que les chemins signalés aient en commun un ou plusieurs événements (dans le cas où NB_FRAMES est supérieur à 2).

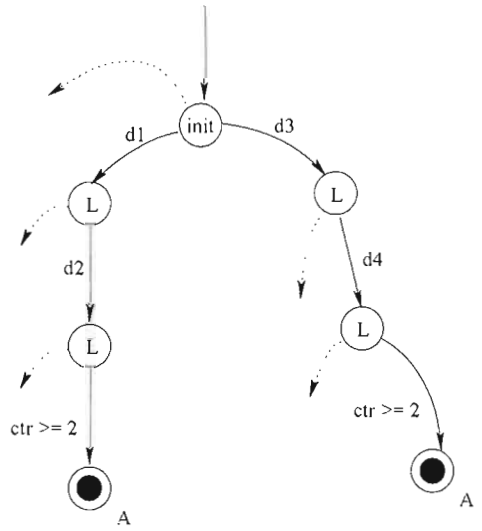

Figure 5.8 Simulation de l'automate de la figure 5.6 modifié à l'aide du mot clé `synchronize`.

La création d'une instance de règle n'est nécessaire que lorsqu'il s'agit d'une nouvelle attaque (un nouvel attaquant ou une nouvelle victime) ou lorsqu'il n'existe aucune instance de règle

active qui traite l'attaque en cours. Pour cela, il faudrait éviter de créer une nouvelle instance de règle tant qu'il y en a une qui est encore active, quand il s'agit du même couple d'adresses source et destination impliquées dans l'attaque. Orchids propose le mot clé `synchronize` qu'on ajoute juste après le nom d'une règle suivi des noms des variables sur lesquels on veut limiter la création d'instances, dans notre cas l'adresse source `$addr_src` et l'adresse destination `$addr_dst`. Ainsi, le résultat de la simulation précédente correspondrait à la figure 5.8. Donc, dès que la première instance de règle se termine, une nouvelle instance de règle, commençant par l'événement d_3, est créée.

5.3.1.4 Solution proposée à l'aide du mot clé cut

Maintenant, pour observer un temps d'attente après chaque alerte, on doit retarder la libération de l'instance de règle. En d'autres termes, l'état d'alerte A ne doit plus être un état terminal. Pour cela, on rajoute une transition partant de l'état A vers un nouvel état terminal. Cette transition doit être franchie après l'écoulement d'un temps d'attente de `WAITIME` secondes (voir figure 5.9).

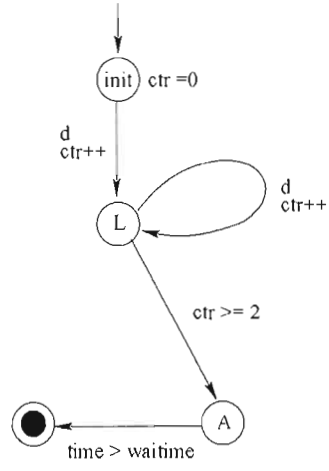

Figure 5.9 Automate modifié de la figure 5.6 prenant en considération un temps d'attente après l'état d'alerte.

Cela dit, le fait de maintenir l'instance de règle active durant un certain temps n'est pas sans conséquences. En effet, les *threads* des états précédant l'état d'alerte restent actifs tant qu'on n'a pas atteint un état terminal. Ceci veut dire que, tant qu'on ne libère pas l'instance de règle, plusieurs *threads* vont continuer à chercher un chemin vers un état terminal et vont

potentiellement atteindre l'état d'alerte (voir figure 5.10). Donc, non seulement le problème de multiplication d'alertes se trouve aggravé mais en plus on assiste à une explosion combinatoire plus ou moins grave selon la valeur donnée à la constante `WAITIME`.

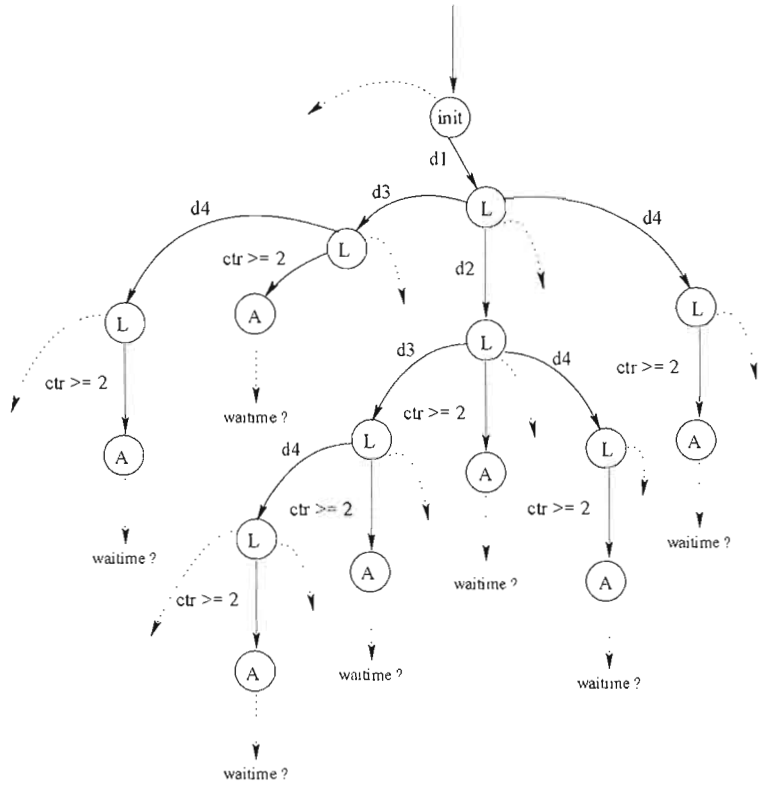

Figure 5.10 Simulation de l'automate de la figure 5.9 modifié à l'aide du mot clé `synchronize`.

Pour empêcher cette dégringolade, Orchids propose la fonction `cut`. Cette fonction met en œuvre le concept de «coupe-choix» de Prolog (*red cut*), donc elle permet d'appliquer un choix définitif *(commited choice)* en arrêtant la recherche de nouveaux chemins à partir d'un état donné. Un appel à `cut("deauth_loop")` à l'entrée de l'état `deauth_alert` permet ainsi d'arrêter tous les *threads* en attente dans le sous-arbre ayant pour racine la première instance d'état `deauth_loop`.

Pourquoi `cut("deauth_loop")` et non pas `cut("init")` ? Tout simplement parce que `cut("init")` empêcherait la création de toute nouvelle instance de règle, même si on est en présence d'une nouvelle attaque (une nouvelle victime ou un nouvel intrus). Le code complet de la signature d'attaque *deauthentication flooding* est présenté à la figure 5.11.

```
/* Fichier deauth_flood.rule */

#define NB_FRAMES   5
#define DELAY       10
#define WAITIME     15

rule deauth_flood
  synchronize($addr_src, $addr_dst)
{

  state init {

    $addr_src = .wifi.sa ;
    $addr_dst = .wifi.da ;

    $counter = 0 ;
    $start_time = .wifi.time ;
    $end_time = .wifi.time + DELAY ;

    if ( .wifi.subtype == "deauth" )
      goto deauth_loop;

  } /* end init */

  state deauth_loop {

    $counter = $counter + 1;

    if (    .wifi.subtype == "deauth"
         && $addr_src == .wifi.sa
         && $addr_dst == .wifi.da
         && .wifi.time < $end_time
         && $counter < NB_FRAMES )
      goto deauth_loop;

    if (    .wifi.subtype == "deauth"
         && $addr_src == .wifi.sa
         && $addr_dst == .wifi.da
         && .wifi.time < $end_time
         && $counter >= NB_FRAMES )
      goto deauth_alert;

    if (    .wifi.subtype == "deauth"
         && $addr_src == .wifi.sa
         && $addr_dst == .wifi.da
         && .wifi.time >= $end_time )
      goto normal_end;

  } /* end deauth_loop */

  state deauth_alert {

    cut("deauth_loop");
    $end_time = .wifi.time + WAITIME ;

    $msg =   "deauthentication flood src:"
           + .wifi.sa + " dst:" + .wifi.da ;

    print ( $msg );

    if ( .wifi.time >= $end_time )
      goto normal_end;

  } /* end deauth_alert */

  state normal_end {

  } /* end normal_end */

} /* end rule deauth_flood */
```

Figure 5.11 Code complet de la signature d'attaque *deauthentication flooding*.

5.3.2 Identification de l'attaque *ChopChop*

Cette attaque a été illustrée à la section 1.3.3 «Attaques contre la confidentialité». On y a vu que pour découvrir une trame chiffrée de N octets, un intrus doit effectuer N cycles de devinettes. Durant chaque cycle, de 1 à 256 trames de même taille et transportant exactement les mêmes données (seul l'ICV est différent) sont envoyées vers un point d'accès. En passant d'un cycle à un autre, la taille des données est réduite d'un octet (voir figure 5.12).

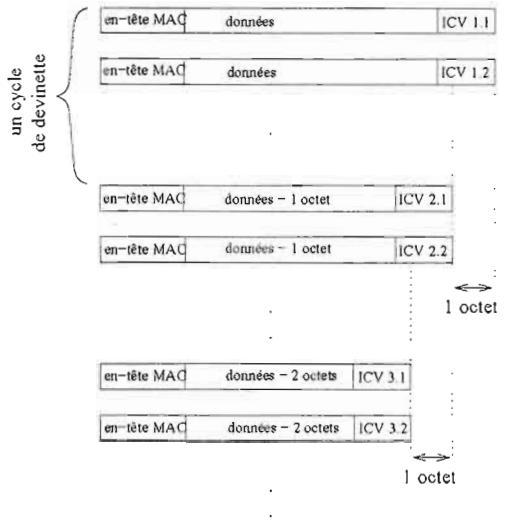

Figure 5.12 Cycles de devinette d'une attaque *ChopChop*.

On vient de décrire l'essentiel de l'attaque, mais d'autres indices peuvent venir préciser cette description, comme par exemple le fait que les trames soient envoyées vers des adresses de destination *multicast* (débutant par `ff`) successivement différentes au niveau d'un seul octet (exemple : `ff:c1:24:e3` et `ff:c1:24:e9`).

La figure 5.13 présente un automate correspondant à cette description de l'attaque *ChopChop*. On commence donc par initialiser un compteur `ctr` à 0 au niveau de l'état `init`. Ensuite, dès qu'on reçoit une trame chiffrée destinée à une adresse *multicast* on passe à l'état `first_loop` en sauvegardant la valeur de la taille de la trame dans la variable bodylen (sur la figure, la taille sauvegardée est N). Le passage à l'état `first_loop` marque le début d'un éventuel cycle de devinette. On s'attend donc à recevoir de 1 à 256 trames de même taille : bodylen' == bodylen (ici `bodylen'` correspond à la taille d'une nouvelle trame alors que `bodylen` représente la taille d'une première trame reçue).

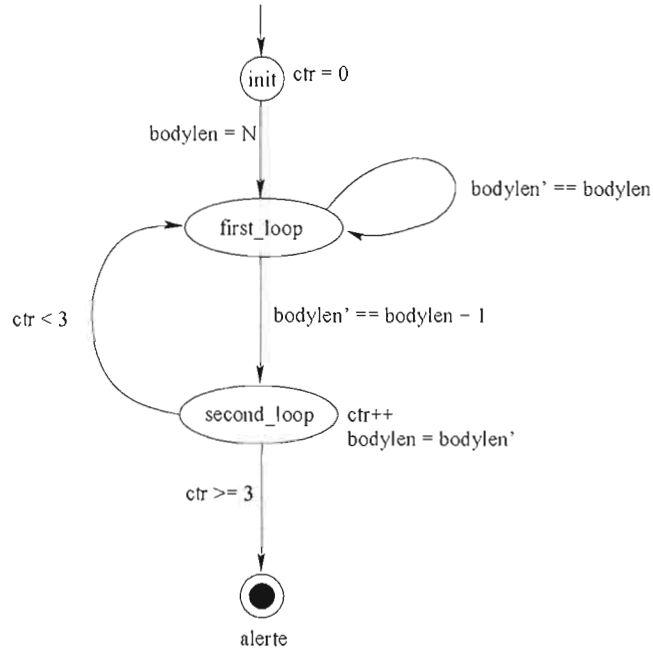

Figure 5.13 Automate correspondant à la signature d'attaque *ChopChop*.

La transition vers l'état `second_loop` marque la fin d'un cycle de devinette et le début du cycle suivant puisque la taille des données est réduite d'un octet : bodylen' == bodylen - 1. Au niveau de l'état `second_loop`, on réinitialise la valeur de la variable bodylen avec celle de bodylen' et on incrémente le compteur `ctr`. Ce compteur sert à compter le nombre de cycles observés. Si on atteint un certain nombre de cycles à partir duquel on peut juger que le comportement observé est fort probablement une attaque *ChopChop* (dans l'exemple on a choisi de le fixer à 3), on lance une alerte, sinon on revient à l'état `first_loop` et on cherche un nouveau cycle, et ainsi de suite.

La figure 5.14 présente une simulation de l'automate de la figure 5.13 sur une séquence d'événements correspondant à quatre cycles de devinette de l'attaque *ChopChop*. On suppose ici que l'attaquant réussit à deviner chaque octet *après la deuxième tentative*. Normalement, à l'observation de la première trame de taille N-3, on devrait lancer une alerte. Cela dit, comme on peut le remarquer sur la figure, même avec un petit exemple, notre espace d'états — donc l'espace mémoire requis — explose rapidement. L'explosion combinatoire serait plus importante avec des cycles plus nombreux et plus longs.

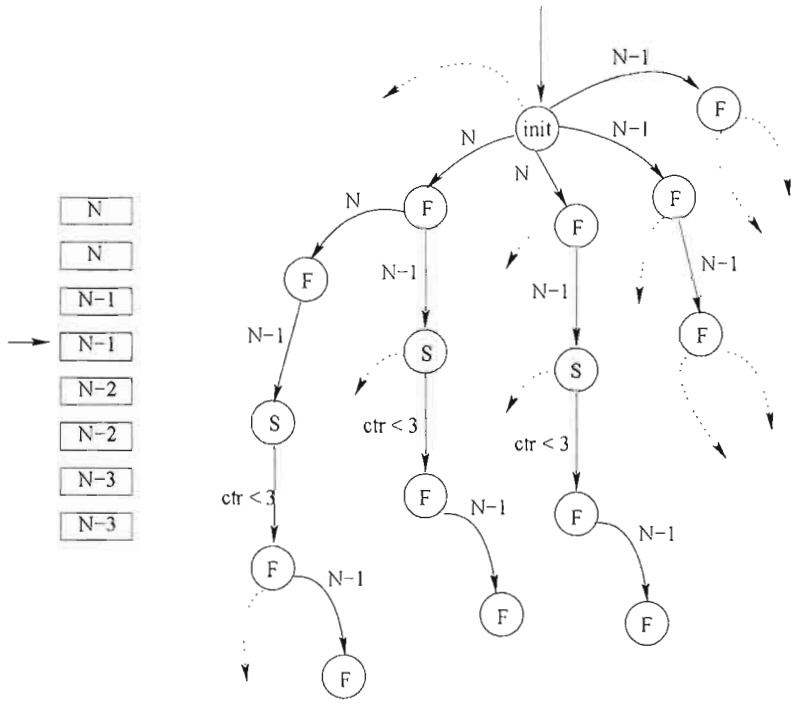

Figure 5.14 Simulation de l'automate de la figure 5.13.

On peut réduire de façon considérable l'explosion combinatoire en utilisant le mot clé `synchronize` sur la variable `bodylen`. En effet, il n'est pas nécessaire de créer une nouvelle instance de règle durant un même cycle de devinette. La figure 5.15 présente la simulation de l'automate de la figure 5.13 sur le même flux d'événements présenté à la figure 5.14 après la synchronisation sur `bodylen`. En comparant les figures 5.14 et 5.15, on peut se rendre compte que dans le premier cas, on crée une instance de règle pour chaque événement et dans le deuxième cas, on crée une instance de règle seulement pour le premier événement d'un même cycle, ce qui représente une amélioration considérable mais pas suffisante dans un scénario plus réaliste.

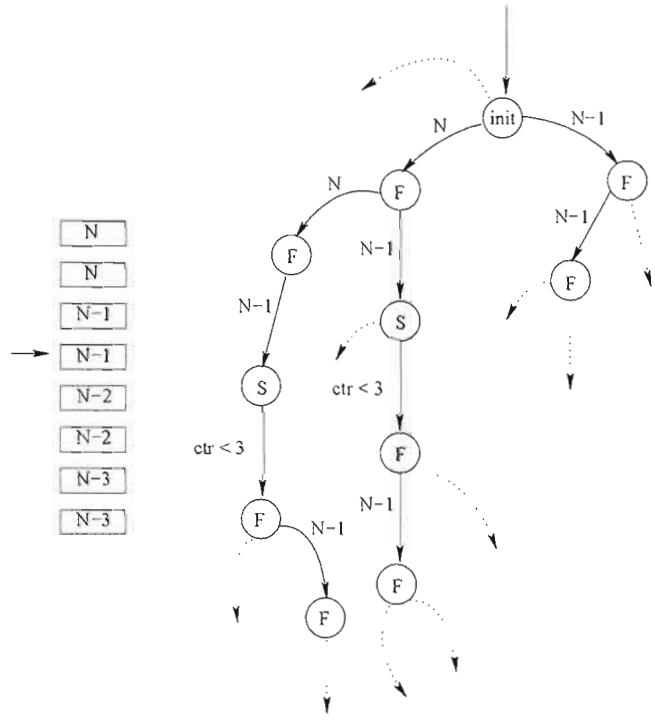

Figure 5.15 Simulation de l'automate de la figure 5.13 modifié à l'aide de synchronize.

En voulant améliorer davantage notre solution au problème de l'explosion combinatoire qu'on vient de voir, on a songé à effectuer un choix définitif *(commited choice)* après chaque cycle en faisant appel à `cut("init")` à l'entrée de l'état second_loop. Bien qu'on réussisse à résoudre le problème d'explosion combinatoire, l'opération est dangereuse. La figure 5.16 présente le résultat de la simulation : comme on peut le remarquer, on atteint l'état d'alerte sans remplir l'espace mémoire sauf qu'à partir du premier `cut("init")` aucune nouvelle instance de règle ne peut être créée. Ceci a pour conséquence la possibilité de masquer des attaques. En

effet, si une nouvelle attaque est lancée entre un `cut("init")` et l'état d'alerte de l'attaque en cours, aucune nouvelle instance de règle ne sera créée pour la nouvelle attaque et elle sera par conséquent cachée au yeux du détecteur d'intrusions.

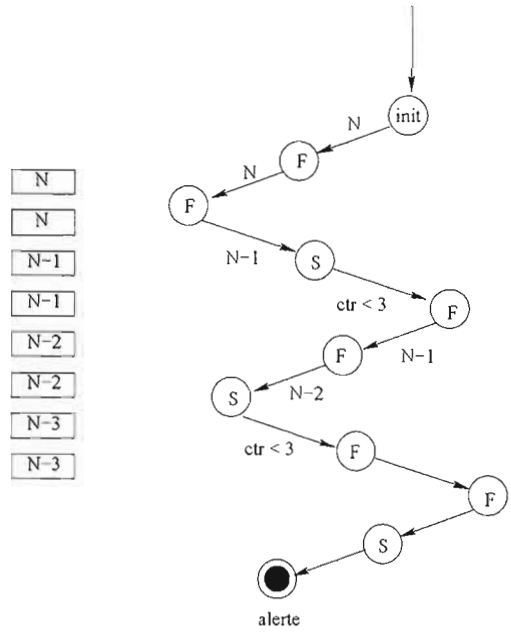

Figure 5.16 Simulation de l'automate de la figure 5.13 modifié à l'aide de la fonction cut.

On arrive donc, avec cet exemple, aux limites d'Orchids. Une idée intéressante serait d'étendre le langage des signatures en définissant un nouveau type d'état «volatile» (cette idée est en cours de discussion avec le développeur d'Orchids, Julien Olivain). En fait, on pense qu'on n'a pas besoin d'allouer un espace mémoire pour chaque nouvelle occurence d'un état qui se répète. En d'autres termes, dans la cas où on a une transition qui boucle sur un état, on pourrait allouer un espace mémoire pour un seul état qu'on écrase avec chaque nouvelle occurence. Ainsi, on réduirait de façon considérable l'arbre de recherche d'une signature d'attaque.

CHAPITRE VI

TESTS ET RÉSULTATS

Dans ce chapitre, on présente les résultats d'expériences effectuées sur notre plateforme de tests. L'objectif ici est de vérifier en pratique les résultats de l'étude théorique effectuée dans les chapitres précédents. Dans la première section, on présente l'architecture de la plateforme et la configuration matérielle utilisée. Dans la deuxième section, on présente deux scénarios d'attaques correspondant aux attaques *deauthentication flooding* et *ChopChop* décrites dans le chapitre précédent.

6.1 Plateforme de tests

Notre plateforme de tests consiste en un réseau local sans-fil 802.11 en mode infrastructure connecté à Internet à travers un réseau câblé (voir figure 6.1). Le réseau sans-fil est exploité principalement par les chercheurs et les invités du LATECE (LAboratoire de recherche sur les TEchnologies du Commerce Electronique) qui s'y connectent à l'aide de leurs ordinateurs portables. Plusieurs autres réseaux sans-fil voisins sont joignables à partir du local du laboratoire ; on est donc dans un environnement réaliste et typique.

Pour réaliser des attaques, on a besoin d'une station qui va jouer le rôle de l'intrus et d'une station victime. On a aussi besoin d'un système indépendant pour effectuer la capture, l'analyse et la détection des attaques. Voici les détails de la configuration matérielle et technique des équipements impliqués dans nos tests :

- **Point d'accès :** Un routeur sans-fil Linksys[1] modèle numéro BEFW11S4 mettant en œuvre le standard IEEE 802.11b. Le point d'accès est connecté à Internet à travers un

[1] Une division de la société Cisco Systems, Inc.

réseau câblé. La configuration sans-fil utilise un filtre d'adresses MAC, ne diffuse pas le SSID *(Service Set IDentifier)* et utilise le chiffrement WEP.

- **Station du client légitime :** Un ordinateur sous Windows 2000 équipé d'une carte D-Link modèle DWL-G122 (interface USB) mettant en œuvre le standard 802.11g.
- **Station du détecteur d'intrusions :** Un ordinateur (Intel Pentium III 996.777 Mhz, 512 Mo de RAM) sous Linux (Fedora Core 6, kernel-2.6.20) équipé d'une carte D-Link modèle DWL-G520 (interface PCI, *chipset* Atheros AR5212 802.11abg) mettant en œuvre le standard 802.11g.
- **Station de l'intrus :** Un ordinateur portable sous Linux (Fedora Core 3, kernel-2.6.12) équipé d'une carte Netgear modèle WG511T (interface CardBus, *chipset* Atheros AR5212 802.11abg) mettant en œuvre le standard 802.11g.

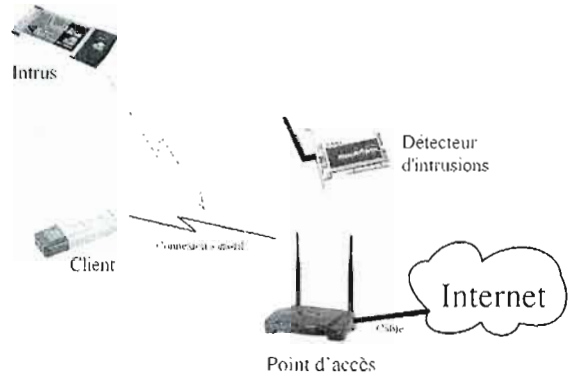

Figure 6.1 Plateforme de tests.

Dans tous nos tests, on suppose que l'intrus n'a aucune connaissance préalable de la configuration du réseau et ne dispose d'aucune information privilégiée.

6.2 Scénarios d'attaques et résultats de la détection

Avant de lancer une attaque, l'intrus a besoin d'identifier sa victime. Pour cela, il doit observer le trafic réseau et choisir la cible qui l'intéresse. Donc, à partir de la station de l'intrus, on active l'interface sans-fil en mode moniteur à l'aide de la commande `wlanconfig`[2] suivante :

[2] `Wlanconfig` est un outil qui vient avec le driver MadWifi (*Multiband Atheros Driver for Wireless Fidelity* — `www.madwifi.org`) de la carte réseau sans-fil.

```
wlanconfig ath0 create wlandev wifi0 wlanmode monitor
```

Ensuite, on effectue une capture, à l'aide de l'outil wireshark[3], sur l'interface `ath0` durant quelques secondes. En analysant les trames, on retrouve les deux adresses MAC correspondant respectivement au point d'accès et à la victime : `00:0F:66:9A:A3:D9` et `00:15:E9:AE:02:B1`. Ces deux adresses sont les seules informations dont l'intrus a besoin.

Pour réaliser les attaques *deauthentication flooding* et *ChopChop*, on a choisi d'utiliser l'outil `aireplay-ng` qui fait partie de la collection d'outils Aircrack-ng[4].

6.2.1 Tests de l'attaque *deauthentication flooding*

Dans ce premier scénario d'attaque, on a lancé sur le poste du client légitime la commande `ping -t` vers un poste situé derrière le point d'accès (un poste dans la zone Internet sur la figure 6.1). L'option `-t` permet d'envoyer, sans interruption, des trames ICMP *request* vers le poste spécifié en paramètre. Si tout se passe bien, on devrait recevoir une trame ICMP *echo* en réponse à chaque requête envoyée. Au cas où un problème survient, le message «`Délai d'attente de la demande dépassé`» est affiché sur le poste du client.

Sur le poste de l'intrus, l'attaque est effectuée en tapant la commande suivante :

```
aireplay-ng -0 10 -a 00:0F:66:9A:A3:D9 -c 00:15:E9:AE:02:B1 ath0
```

Le paramètre `-0` spécifie le type d'attaque qu'on voudrait effectuer (c'est l'attaque *deauthentication flooding* qui est choisie ici). Le chiffre 10, passé en paramètre, indique le nombre de trames désauthentification à envoyer. On peut fixer un nombre supérieur à 10 si on veut prolonger la durée de l'attaque mais ceci n'a pas de conséquences notables sur le déroulement de l'expérience. Seul un changement dans la fréquence d'émission des trames pourrait avoir des conséquences mais l'outil n'offre pas cette possibilité.

Les options `-a` et `-c` permettent de spécifier respectivement l'adresse du point d'accès et celle de la station client. On indique finalement l'interface qu'on veut utiliser pour lancer l'attaque. Ce qui suit est alors affiché sur le poste de l'intrus :

---

[3] `www.wireshark.org`

[4] `http://www.aircrack-ng.org/`

```
07:26:18  Sending DeAuth to station  -- STMAC: [00:15:E9:AE:02:B1]
07:26:18  Sending DeAuth to station  -- STMAC: [00:15:E9:AE:02:B1]
07:26:18  Sending DeAuth to station  -- STMAC: [00:15:E9:AE:02:B1]
07:26:20  Sending DeAuth to station  -- STMAC: [00:15:E9:AE:02:B1]
07:26:20  Sending DeAuth to station  -- STMAC: [00:15:E9:AE:02:B1]
07:26:20  Sending DeAuth to station  -- STMAC: [00:15:E9:AE:02:B1]
07:26:20  Sending DeAuth to station  -- STMAC: [00:15:E9:AE:02:B1]
07:26:20  Sending DeAuth to station  -- STMAC: [00:15:E9:AE:02:B1]
07:26:21  Sending DeAuth to station  -- STMAC: [00:15:E9:AE:02:B1]
07:26:21  Sending DeAuth to station  -- STMAC: [00:15:E9:AE:02:B1]
```

On a l'impression que chaque ligne signale l'émission d'une seule trame de type désauthentification. Même la documentation d'aireplay-ng indique que le nombre de trames émis durant l'attaque est égal au nombre passé en paramètre. Mais en regardant le code source de l'outil, et en faisant une capture à l'aide de l'outil wireshark, on a constaté que le nombre de trames est beaucoup plus élevé en réalité. En fait, chaque ligne affichée signale l'envoie de 22 trames de type désauthentification : 11 trames ayant comme adresse source `00:0F:66:9A:A3:D9` et adresse destination `00:15:E9:AE:02:B1`, suivies par 11 trames ayant pour adresse source `00:15:E9:AE:02:B1` et adresse destination `00:0F:66:9A:A3:D9` (la direction inverse des 11 premières trames).

Lorsque le client légitime reçoit la première trame de type désauthentification, sa connexion est perdue et le message «`Délai d'attente de la demande dépassé`» est affiché. Et après la réception de la dernière trame, la station du client réussit à se réauthentifier auprès du point d'accès et recommence à recevoir les trames ICMP *echo*. On comprend donc que l'attaque a été effectuée avec succès et on s'attend à trouver dans le journal du détecteur d'intrusions, au plus, les deux nouvelles lignes suivantes :

```
FU str[66]: "deauthentication flood src:00:0f:66:9a:a3:d9 dst:00:15:e9:ae:02:b1"
FU str[66]: "deauthentication flood src:00:15:e9:ae:02:b1 dst:00:0f:66:9a:a3:d9"
```

Chaque ligne signalant les premières trames de type désauthentification implique un couple différent d'adresses MAC source et destination. Dans ce qui suit, on va analyser trois cas de figures pour la détection de cette attaque et on va illustrer les conséquences sur les performances et sur le nombre d'alertes signalées dans chaque cas.

Sur le tableau 6.1 on observe les résultats des trois expériences. Chaque ligne du tableau correspond à un cas de figure particulier. Les colonnes du tableau représentent (respectivement de gauche à droite) :

- Le nombre maximum d'instances de règles observées durant l'attaque.
- Le nombre maximum d'instances d'états observées durant l'attaque.
- Le nombre maximum de *threads* actifs observés durant l'attaque.
- La durée de vie des instances de règles (en secondes).
- Le nombre d'alertes consignées dans le journal du détecteur d'intrusions pour cette attaque.

	Nombre maximum d'instances de règles	Nombre maximum d'instances d'états	Nombre maximum de *threads* actifs	Durée de vie des instances de règles	Nombre d'alertes consignées
Signature sans `synchronize` et sans `WAITIME`	10	40	70	13	202
Signature avec `synchronize` et sans `WAITIME`	2	7	12	13	34
Signature avec `synchronize`, `WAITIME` et `cut`	2	7	3	21	2

Tableau 6.1 Résultats du scénario d'attaque *deauthentication flooding*.

Dans le premier cas de figure, on lance Orchids avec une seule signature d'attaque correspondant à la première tentative d'identification effectuée à la section 5.3.1 «Identification de l'attaque de désauthentification par inondation» (sans `synchronize` et sans le temps d'attente `WAITIME`). Le journal du détecteur d'intrusions signale 202 alertes.

Chaque nouvelle trame, émise par l'intrus et arrivée au détecteur d'intrusions, crée une nouvelle instance de règle et une ou plusieurs nouvelles instances d'états. Vu qu'à partir de l'état deauth_loop on peut atteindre trois états, chaque nouvelle instance de l'état deauth_loop crée trois *threads* : deux *threads* restent en attente jusqu'à la fin de la règle (sur un *timeout* ou une alerte) et un *thread* réussit à passer la transition sur laquelle il est bloqué (la réception

d'une trame de type désauthentification), ce qui explique le fait que le nombre de *threads* est à peu près le double du nombre d'états.

Dans le deuxième cas de figure, on lance Orchids avec une seule signature d'attaque correspondant à la deuxième tentative d'identification (avec `synchronize` et sans le temps d'attente `WAITIME`). Le journal du détecteur d'intrusions signale 34 alertes. La deuxième ligne du tableau 6.1 présente les mesures effectuées pour ce cas de figure. On remarque que le nombre maximum d'instances de règles actives est de deux cette fois. En effet, c'est grâce au mot-clé synchronize qu'on a réussi à limiter la création des instances de règles seulement pour un couple d'adresses MAC source et destination différents. Dans cette expérience, on a deux adresses MAC qu'on utilise dans les deux directions, ce qui explique le fait qu'on trouve deux instances de règles. Comme conséquence, le nombre d'instances d'états et de *threads* a aussi été réduit par rapport au premier cas de figure.

Dans le troisième cas de figure, on lance Orchids avec une seule signature d'attaque correspondant à la dernière solution dont le code complet a été présenté à la figure 5.11 (avec `synchronize`, `WAITIME` et `cut`). Le journal du détecteur d'intrusions signale, comme prévu, deux alertes. Sur la troisième ligne du tableau 6.1, on constate deux différences notables par rapport au deuxième cas de figure (signature avec `synchronize` et sans `WAITIME`). La première différence est dans le nombre de *threads*. Dans ce troisième cas de figure, le nombre de *threads* est réduit grâce à la fonction cut. La deuxième différence concerne la durée de vie des instances de règles qui est plus élevée à cause du temps d'attente `WAITIME` observé après chaque alerte.

6.2.2 Tests de l'attaque *ChopChop*

`aireplay-ng` permet de réaliser l'attaque *ChopChop* à l'aide de la commande suivante :

```
aireplay-ng -4 -h 00:15:E9:AE:02:B1 ath0
```

Le paramètre `-4` indique le type d'attaque qu'on veut effectuer (c'est l'attaque *ChopChop* dans ce cas). Ensuite, on précise l'adresse MAC de la station qu'on veut attaquer à l'aide de l'option `-h`.

Le système se met donc en mode écoute et affiche la première trame chiffrée, provenant de la station ayant l'adresse MAC `00:15:E9:AE:02:B1`, observée sur l'interface `ath0`. On choisit ensuite si on veut découvrir le contenu de cette trame ou si on veut continuer à chercher une

autre trame (voir figure 6.2). Lorsqu'on a fixé notre choix sur la trame qu'on veut découvrir, *ChopChop* commence à effectuer les cycles de devinette.

```
Read 5 packets...

        Size: 100, FromDS: 0, ToDS: 1 (WEP)

             BSSID  =  00:0F:66:9A:A3:D9
         Dest. MAC  =  00:0F:66:9A:A3:D9
        Source MAC  =  00:15:E9:AE:02:B1

        0x0000:  0841 0201 000f 669a a3d9 0015 e9ae 02b1  .A....f.........
        0x0010:  000f 669a a3d9 c036 1cb9 5c00 a304 e836  ..f....6..:..6
        0x0020:  6d9a 353f e586 ab99 f905 c451 979c 959b  m.5?.......Q....
        0x0030:  5598 9202 08be 7f69 2705 0fdc 059c cc27  U.....^?i'......'
        0x0040:  aa23 423a 84b5 5687 b39f 0834 69ed 893d  .#B:..V....4i..=
        0x0050:  584e cfa4 124f 6013 1d5f 55e4 d5fd ad3e  XN...O'.._U....>
        0x0060:  a5c8 54a5                                ..T.

Use this packet ?
```

Figure 6.2 Une trame chiffrée sélectionnée par *ChopChop*.

Pour ce scénario d'attaque, on a lancé Orchids avec une seule signature d'attaque correspondant à la première signature décrite à la section 5.3.2 «Identification de l'attaque *ChopChop*» (sans la synchronisation sur la taille des trames). Comme prévu, les mesures donnent des chiffres élevés comparativement à ce qu'on a avec le scénario d'attaque *deauthentication flooding* (voir première ligne du tableau 6.2). En effet, une nouvelle instance de règle est créée avec chaque nouvelle trame reçue, ce qui conduit vers une explosion rapide de l'espace mémoire.

Dans le deuxième cas de figure, on a lancé Orchids avec une signature d'attaque correspondant à la signature décrite à l'aide des mots-clé cut et synchronize. La deuxième ligne du tableau 6.2 présente les mesures effectuées sur le détecteur d'intrusions pour ce deuxième cas de figure. Comme on peut le remarquer, une seule instance de règle est active durant l'attaque, ceci à cause du `cut("init")` effectué à l'entrée de l'état second_loop qui empêche la création de toute nouvelle instance de règle tant que celle-ci reste active. Après chaque trois cycles de devinette[5], une nouvelle alerte est consignée dans le journal du détecteur d'intrusions.

[5] Ce choix a été fixé au chapitre précédent lors de l'identification de l'attaque *ChopChop*.

	Nombre maximum d'instances de règles	Nombre maximum d'instances d'états	Nombre maximum de *threads* actifs
Signature sans cut et sans synchronize	258	95582	326740
Signature avec cut et synchronize	1	9628	16

Tableau 6.2 Résultats du scénario d'attaque *ChopChop*.

Avec ces différentes expériences on a pu confirmer les résultats de l'étude théorique effectuée dans les chapitres précédents. On a vu, par exemple, que les performances du système de détection d'intrusions peuvent varier de façon considérable selon les choix effectués pour écrire les signatures d'attaques.

On constate aussi qu'en ce qui concerne l'attaque *ChopChop*, même avec la meilleure signature d'attaque qu'on a pu écrire, les performances (en consommation d'espace mémoire et en nombre d'alertes) sont faibles par rapport à ce qu'on trouve avec l'attaque *deauthentication flooding*. Ceci rejoint l'idée avancée au chapitre précédent proposant d'étendre le langage des signatures d'attaques pour pouvoir identifier correctement les attaques basées sur des événements répétitifs (en boucle) semblables à l'attaque *ChopChop*.

CONCLUSION

Jean Goubault-Larrecq et Muriel Roger ont suggéré que «la détection d'intrusions complexes est essentiellement une activité de *model-checking* de formules temporelles (décrivant des classes de scénarios d'attaques) sur des flux d'événements»[6]. Cela dit, nous avons montré dans ce mémoire de maîtrise les limites de l'algorithme proposé dans l'article «*Log auditing through model checking*» [26] (et mis en œuvre dans l'outil Orchids [23]) pour détecter efficacement les attaques dans un environnement sans-fil, notamment l'attaque *ChopChop*.

Dans le cadre du travail décrit dans ce mémoire, nous avons tout d'abord développé et intégré, dans Orchids, notre propre module de capture et de découpage des trames 802.11. Par la suite, nous avons décrit, à l'aide de signatures, un certain nombre d'attaques, notamment, *ChopChop* — à notre connaissance, nous somme les premiers à détecter cette attaque —, *ARP Replay*, et la *deauthentication flooding*. Ces attaques ont ensuite été mises en œuvre, puis détectées avec succès dans un environnement réel (trois machines : un client, un attaquant et le détecteur d'intrusions, plus un point d'accès).

De façon générale, les nouveaux systèmes lancent de nouveaux défis aux détecteurs d'intrusions (en plus des défis de base comme la prise en charge de l'aspect temporel, multiplication des alertes, expressivité des langages de signatures, etc.). Ces derniers tentent d'appliquer des approches universelles (dans leurs contextes initiaux) sans prendre en compte la spécificité des systèmes surveillés.

Nous croyons qu'il est important de prendre du recul par rapport aux choix initiaux et de considérer les nouveaux systèmes et les nouveaux défis lors de l'élaboration des outils. En d'autres termes, il faudrait repenser au problème de la détection d'intrusions dans un cadre plus spécifique (moins large).

Dans le cas des réseaux sans-fil, à notre connaissance, il n'y a pas eu un grand effort d'analyse et d'étude dans ce sens pour développer des solutions appropriées à ce type de réseau (incluant notre solution). La majorité des solutions qu'on trouve sont soit des extensions à des outils conçus initialement pour les réseaux filaires, soit des solutions bâties sur une approche mise en œuvre par ces outils.

---

[6] `http://www.liafa.jussieu.fr/web9/manifsem/description_fr.php?idcongres=602`

Plusieurs pistes de travaux futurs peuvent être envisagées. On pense notamment à la possibilité d'étendre le langage des signatures pour permettre une identification plus efficace d'une série d'événements répétitifs ou d'améliorer plutôt le processus d'analyse en gardant le même langage. Une autre idée consisterait à revoir le problème de la détection d'intrusions en prenant en considération la particularité du système ciblé.

BIBLIOGRAPHIE

[1] W. A. Arbaugh, N. Shankar, and J. Wang. Your 802.11 Network has no Clothes. In *Proceedings of the First IEEE International Conference on Wireless LANs and Home Networks*, pages 131–144, December 2001.

[2] J. Bellardo and S. Savage. 802.11 denial-of-service attacks: Real vulnerabilities and practical solutions. In *Proceedings of the Twelfth USENIX Security Symposium*, pages 15–28, Washington, DC, USA, August 2003. USENIX Association.

[3] C. Bidan, G. Hiet, L. Mé, B. Morin, and J. Zimmermann. Vers une détection d'intrusions à fiabilité et pertinence prouvables, octobre 2006. `http://www.rennes.supelec.fr/rennes/si/equipe/lme/PUBLI/Bal.06.pdf`.

[4] A. Bittau, M. Handley, and J. Lackey. The final nail in WEP's coffin. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 386–400, Washington, DC, USA, 2006. IEEE Computer Society.

[5] L. Butti. Détection d'Intrusion dans les Réseaux 802.11. In *Symposium sur la Sécurité des Technologies de l'Information et des Communications 2006*, pages 411–433. École Supérieure et d'Application des Transmissions, 2006.

[6] B. Caswell and J. Hewlett. *Snort Users Manual*, 2.6.1 edition, December 2006. Disponible sur la page Web `http://www.snort.org/docs/snort_manual/2.6.1/snort_manual.pdf`.

[7] E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic model checking. In *Computer Aided Verification*, pages 419–422. Springer-Verlag, LNCS-1102, 1996.

[8] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications: A practical approach. *Tenth ACM Symposium on Principles of Programming Languages*, pages 244–263, January 1983. `http://www.cs.cmu.edu/~emc/papers.htm`.

[9] M. Gast. *802.11 Wireless Networks : The Definitive Guide*. O'Reilly, April 2002.

[10] J. Goubault-Larrecq. Un algorithme pour l'analyse de logs. Rapport de recherche LSV-02-18, Laboratoire Spécification et Vérification, ENS Cachan, France, novembre 2002.

[11] J. Goubault-Larrecq, J.-P. Pouzol, S. Demri, L. Mé, and P. Carle. Langages de détection d'attaques par signatures. Sous-projet 3, livrable 1 du projet RNTL DICO. Version 1, juin 2002.

[12] F. Guo and T.-C. Chiueh. Sequence number-based MAC address spoof detection. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID'05)*, pages 309–329, Seattle, WA, USA, September 2005.

[13] G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[14] M. Hurfin, J.-P. Le Narzul, F. Majorczyk, L. Mé, A. Saidane, E. Totel, and F. Tronel. A dependable intrusion detection architecture based on agreement services. In *Proc. of*

the 8th International Symposium on Stabilization Safety and Security (SSS 2006), LNCS 4280, pages 378–394, Dallas, Texas, Nov 2006. Springer-Verlag. `http://www.rennes.supelec.fr/rennes/si/equipe/lme/PUBLI/Hal.06.pdf`.

[15] M. Kershaw. *Kismet 2005-08-R1*, August 2005. Disponible sur la page Web du projet Kismet, `http://www.kismetwireless.net`.

[16] LAN/MAN Standards Committee of the IEEE Computer Society. *Information technology - Telecommunications and Information Exchange between Systems - Local and Metropolitan Area Networks - Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.* IEEE Standards Board, ANSI/IEEE Std 802.11 (R2003) edition, 1999. Reaffirmed 12 June 2003.

[17] M. Li, M. Claypool, and R. Kinicki. Wireless sniffing by example — how to build and use an IEEE 802.11 wireless network sniffer. Technical Report WPI-CS-TR-05-19, Department of Computer Science at Worcester Polytechnic Institute, November 2005.

[18] R. Mateescu. *Vérification des propriétés temporelles des programmes parallèles.* Thèse de doctorat, Institut National Polytechnique de Grenoble, avril 1998.

[19] B. Morin. *Corrélation d'alertes issues d'outils de détection d'intrusions avec prise en compte d'informations sur le système surveillé.* Thèse de doctorat, Institut National des Sciences Appliquées de Rennes, fèvrier 2004.

[20] R. Neumerkel and S. Gross. A Sophisticated Solution for Revealing Attacks on Wireless LAN. In *Proceedings of the 3rd International Conference on Trust, Privacy and Security in Digital Business (TrustBus'06)*, volume 4083 of *Lecture Notes in Computer Science*, pages 223–232, Krakow, Poland, September 2006. Springer-Verlag.

[21] M. Norton and D. Roelker. Snort 2.0 Hi-Performance Multi-rule Inspection Engine, February 2003.

[22] J. Olivain. Plate-forme de détection d'intrusions : Analyse et corrélation temporelle d'événements en temps réel. Rapport de stage, Laboratoire Spécification et Vérification, ENS Cachan, France, novembre, décembre 2003.

[23] J. Olivain and J. Goubault-Larrecq. The Orchids intrusion detection tool. In K. Etessami and S. Rajamani, editors, *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05)*, volume 3576 of *Lecture Notes in Computer Science*, pages 286–290, Edinburgh, Scotland, UK, July 2005. Springer-Verlag.

[24] Passeport Canada. Projet d'identification faciale: Rapport de l'évaluation des facteurs relatifs à la vie privée. Site Web officiel de Passeport Canada (www.ppt.gc.ca), mai 2006.

[25] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In USENIX, editor, *Seventh USENIX Security Symposium*, pages 31–52, San Antonio, Texas, January 1998. USENIX.

[26] M. Roger and J. Goubault-Larrecq. Log auditing through model checking. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 220–236, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society Press.

[27] M. Roger and J. Goubault-Larrecq. Log auditing through model-checking. Technical Report LSV-01-3, Laboratoire Spécification et Vérification, ENS de Cachan, France, April 2001.

[28] M. Salmanian, S. Leonard, J. H. Lefebvre, and S. Knight. Intrusion detection in 802.11 wireless local area networks. Technical Memorandum DRDC Ottawa TM 2004-120, Defence R&D Canada, Ottawa, Ontario, September 2004.

[29] P. Schnoebelen, B. Bérard, M. Bidoit, F. Laroussinie, and A. Petit. *Vérification de logiciels : techniques et outils du model-checking.* Vuibert, avril 1999.

[30] E. Tews, R.-P. Weinmann, and A. Pyshkin. Breaking 104 bit WEP in less than 60 seconds. Cryptology ePrint Archive, Report 2007/120, April 2007.

[31] F. Valeur. *Real-Time Intrusion Detection Alert Correlation.* PhD thesis, University of California, Santa Barbara, May 2006.

[32] A. Vladimirov, K. V. Gravrilenko, and A. A. Mikhailovskiy. *Wi-Foo: Piratage et défense des réseaux sans fil.* CampusPress, juin 2005.

[33] P. Wolper. Constructing automata from temporal logic formulas: a tutorial. In *Lectures on formal methods and performance analysis: first EEF/Euro summer school on trends in computer science*, pages 261–277, New York, NY, USA, 2002. Springer-Verlag New York, Inc.

[34] J. Wright. Detecting Wireless LAN MAC Address Spoofing, January 2003. `http://forskningsnett.uninett.no/wlan/download/wlan-mac-spoof.pdf`. Consulté en novembre 2007.

[35] J. Zimmermann and L. Mé. Les systèmes de détection d'intrusions : principes algorithmiques, juin 2002. `http://www.rennes.supelec.fr/rennes/si/equipe/lme/PUBLI/ZM02.pdf`.