Architecture expérimentale pour la détection d'intrusions dans un système informatique

Philippe Biondi
philippe.biondi@webmotion.com

Avril-Septembre 2001

Résumé

La détection d'intrusions consiste à découvrir ou identifier l'utilisation d'un système informatique à d'autres fins que celles prévues. C'est une technique à multiples facettes, difficile à cerner lorsqu'on ne les manipule pas. Mais la plupart des travaux effectués dans ce domaine restent difficiles à comparer. On peut rarement mettre deux modèles sur un pied d'égalité, et il est peu aisé de mettre à l'épreuve plusieurs modèles, ou encore d'en développer d'autres radicalement différents sans tout reconstruire.

La détection d'intrusion sera placée parmi les autres techniques anti-intrusion. Ensuite, un approfondissement sera effectué sur cette technique. Plusieurs domaines comme les réseaux neuronaux, les sytèmes multi-agents ou encore le data-mining seront rapprochés de cette technique. Enfin, quelques modèles, prototypes ou produits de détection d'intrusions seront présentés.

Dans une deuxième partie, l'architecture EIDF (Experimental Intrusion Detection Framework) sera introduite. Sa modularité sera mise en avant pour répondre au problème. Plusieurs modèles de détection d'intrusion seront ensuite présentés et implémentés. L'architecture sera enfin mise à profit pour les tester.

Table des matières

Table des figures

Liste des algorithmes

Listings

Introduction

Contexte

Le concept de système de détection d'intrusions a été introduit en 1980 par James Anderson [And80]. Mais le sujet n'a pas eu beaucoup de succés. Il a fallu attendre la publication d'un modèle de détection d'intrusions par Denning en 1987 [Den87] pour marquer réellement le départ du domaine. En 1988, il existait au moins trois prototypes : Haystack [Sma88] (voir section 4.1), NIDX [BK88], et [SSHW88].

La recherche dans le domaine s'est ensuite développée, le nombre de prototypes s'est enormément accru. Le gouvernement des États-Unis a investi des millions de dollars dans ce type de recherches dans le but d'accroître la sécurité de ses machines. La détection d'intrusion est devenue une industrie mature et une technologie éprouvée : à peu près tous les problèmes simples ont été résolus, et aucune grande avancée n'a été effectuée dans ce domaine ces dernières années, les éditeurs de logiciels se concentrant plus à perfectionner les techniques de détection existantes.

Quelques voies restent cependant relativement inexplorées :
- les mécanismes de réponse aux attaques,
- les architectures pour les sytèmes de détection d'intrusions distribués,
- les standards d'inter-opérabilité entre différents systèmes de détection d'intrusion,
- la recherche de nouveaux paradigmes pour effectuer la détection d'intrusion.

Motivations

Une des approches de la sécurité informatique est de créer un système complètement sécurisé, c'est la prévention. Mais il est très rarement possible de rendre un système complètement inattaquable pour plusieurs raisons.

- La plupart des systèmes informatiques ont des failles de sécurité qui les rendent vulnérables aux intrusions. Les trouver et les réparer toutes n'est pas possible pour des raisons techniques et économiques.
- Les systèmes existants ayant des failles connues ne sont pas facilement remplacés par des systèmes plus surs, principalement parce qu'ils ont des fonctionnalités intéressantes que n'ont pas les systèmes plus surs, ou parce qu'ils ne peuvent pas être remplacés pour des raisons économiques.
- Déployer des sytèmes sans failles est très dur voire impossible car des failles

sont inconnues ou inévitables
– Même les systèmes les plus sûrs sont vulnérables aux abus de la part d'utilisateurs légitimes qui profitent de leurs privilèges. ou souffrent de la négligence des règles de sécurité par ceux-ci.

En réponse à ces difficultés pour développer des systèmes sécurisés, un nouveau modèle de gestion de la sécurité des systèmes a émergé [Bac99] (figure 1).

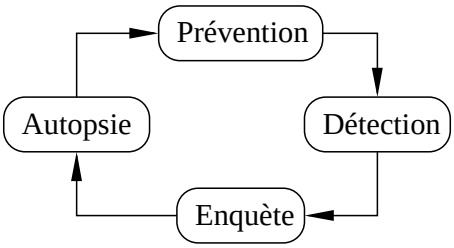

FIG. 1 – Un modèle de gestion de la sécurité d'un système informatique

Dans cette approche plus réaliste, la prévention n'est qu'une des quatres parties de la gestion de la sécurité. La partie *détection* est à la recherche de l'exploitation de nouvelles brèches. La partie *enquète* essaye de déterminer ce qui est arrivé, en s'appuyant sur les informations fournies par la partie *détection*. La partie *autopsie* consiste à chercher comment empêcher des intrusions similaires dans le futur.

Par le passé quasiment toute l'attention des chercheurs s'est portée sur la partie *prévention*. La *détection* est maintenant beaucoup prise en compte, mais les deux autres parties ne reçoivent pas encore toute l'attention qu'elles méritent.

Nous nous intéresserons plus particulièrement ici à la partie *détection*.

Définitions

Tout d'abord, quelques définitions :

Système informatique : nous appellerons système informatique une ou plusieurs machines mises à la disposition de zéro, un ou plusieurs utilisateurs légitimes pour toutes sortes de tâches.

Intrusion : nous appellerons intrusion toute utilisation d'un système informatique à des fins autres que celles prévues, généralement dues à l'acquisition de privilèges de façon illégitime. L'intrus est généralement vu comme une personne étrangère au système informatique qui a réussi à en prendre le contrôle, mais les statistiques montrent que les utilisations abusives (du détournement de ressources à l'espionnage industriel) proviennent le plus fréquement de personnes internes ayant déjà un accés au système.

Mécanisme d'audit : nous appellerons mécanisme d'audit toute partie de code du système informatique dont le but est de reporter des informations sur les opérations qu'il lui est demandé d'accomplir.

Journal d'audit : nous appellerons journal d'audit l'ensemble des informations générées par les mécanismes d'audit.

Événement : étant donné un niveau de granularité, nous appellerons événement toute opération élémentaire. Par extension, nous appellerons également événement le report de celui-ci par un mécanisme d'audit

Flux d'audit élémentaire : nous appellerons flux d'audit élémentaire une suite temporelle reportant les événements se produisant sur une même partie du système et traduisant son comportement.

Flux d'audit : nous appellerons flux d'audit une suite temporelle d'événements, pouvant être l'entremêlement de plusieurs flux élémentaires.

Détection d'intrusions : la détection d'intrusions consiste à analyser les informations collectées par les mécanismes d'audit de sécurité, à la recherche d'éventuelles attaques. Bien qu'il soit possible d'étendre le principe, nous nous concentrerons sur les systèmes informatiques. Les méthodes de détection d'intrusion diffèrent sur la manière d'analyser le journal d'audits.

Première partie

État de l’art

Chapitre 1

Taxonomie des techniques anti-intrusion

La détection d'intrusions a longtemps été vue comme le moyen le plus prometteur de combattre les intrusions. Pourtant elle fait partie d'un ensemble plus grand de techniques anti-intrusion qui ont aussi leur place.

[HB95] propose une taxonomie de toutes ces techniques, que l'on peut résumer en le schéma de la figure 1.1, et que l'on détaillera ci-après.

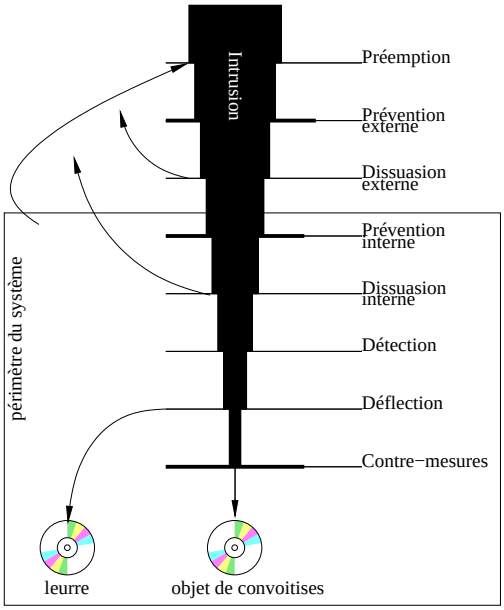

FIG. 1.1 – Résumé des différentes techniques anti-intrusion

Pour chaque technique anti-intrusion, une analogie également tirée de [HB95], sera faite dans le monde réel, en prenant comme sujet quelqu'un qui se promène dans la rue

et qui ne veut pas se faire voler son portefeuille.

1.1 Préemption

Les techniques de préemption d'intrusion prennent place avant d'éventuelles tentatives d'intrusion pour les rendre moins susceptibles de se dérouler effectivement. Cela revient à frapper l'autre avant qu'il ne le fasse. C'est souvent une approche dangereuse où des innocents peuvent être touchés. On peut inclure l'éducation des utilisateurs, la prise de mesures préventives à l'encontre des usagers commençant à se montrer enclins à outrepasser leurs droits, l'infiltration de communautés de pirates pour se tenir au courant des techniques et motivations de ceux-ci.

Analogie : *Sympathiser avec un pickpocket et éliminer quiconque fera allusion à notre portefeuille* Ce n'est pas très ciblé et cela peut frapper des innocents.

Mise en œuvre :

Veille active : on tente d'empêcher de futures intrusions en cherchant les signes préliminaires d'une activité non désirée, en remarquant par exemple les étapes exploratoires d'une intrusion, et en prenant sans attendre des mesures contre les utilisateurs concernés. On peut aussi récompenser les utilisateurs qui trouvent des failles ou des usages non-autorisés.

Infiltration : ce sont les efforts des personnes en charge de la sécurité des systèmes pour acquérir des informations plus officieuses pour complémenter les alertes officielles d'organismes comme le Computer Emergency Response Team (CERT) ou autres rapports de vulnérabilités. Une infiltration plus insidieuse pourrait être d'inonder les réseaux pirates d'informations fausses pour embrouiller ou dissuader ceux-ci.

1.2 Prévention

Les techniques de prévention, mises en place à l'intérieur ou à l'extérieur du système, consistent à concevoir, implémenter, configurer le système assez correctement pour que les intrusions ne puissent avoir lieu, ou au moins soient sévèrement gênées.

On peut par exemple choisir d'isoler une machine dans une cage ou de ne pas connecter une machine au réseau et éviter ainsi toute intrusion venant de là[1].

Idéalement, cette technique pourrait empêcher toute attaque et pourrait être considérée comme la technique anti-intrusion la plus forte. Dans un monde réel, cette technique est irréalisable, il reste toujours quelques failles.

Analogie : *Engager des gardes du corps musclés et éviter les mauvais voisinages.* C'est une approche de choix lorsqu'elle marche mais elle coûte cher, elle n'est pas com-

[1] il reste bien sur à protéger l'accès physique à la machine, à contrôler les émissions radio-électriques par exemple en utilisant des cages de Faraday, à n'utiliser que du personnel de confiance, ...

mode d'utilisation, et n'est pas totalement fiable. Un garde du corps peut être distrait ou battu par un malfaiteur ou mis en défaut par des moyens inattendus ou nouveaux.

Mise en œuvre :

Conception et implémentation correcte : c'est l'utilisation des techniques classiques d'identification, d'authentification, de contrôle d'accès physique et logique. On peut trouver des méthodologies comme comme l'INFOSEC Assesslent Methodology (IAM) (http://www.nsa.gov/isso/iam/index.htm) ou encore la rfc 2196 (http://www.ietf.org/rfc/rfc2196.txt). Les techniques sont bien connues mais souvent peu commodes et chères.

Outils de recherche de failles : qu'ils soient lancés à la main ou automatiquement, ils recherchent un grand éventail d'irrégularité : logiciels non-autorisés, mots de passe faibles, permissions d'accès ou propriété inapropriées, nœuds fantômes sur le réseau, ...

Pare-feux : ils examinent le flux d'information entre deux réseau et les protègent l'un de l'autre en bloquant une partie du trafic.

1.3 Dissuasion

Les techniques de dissuasion tentent de réduire la valeur apparente d'une intrusion et de faire percevoir les efforts nécessaires à l'intrusion et les risque encourus plus grands que ce qu'ils ne sont. La dissuasion encourage un cyber-criminel à s'intéresser à d'autre systèmes promettant plus de bénéfices à moindre coût. Cette approche inclus la dévaluation du système par camouflage et l'augmentation du risque perçu par l'affichage de messages de mises en garde, ou qui surestiment la surveillance du système. Le but final étant de faire croire que le jeu n'en vaut pas la chandelle.

Analogie : *Se dévêtir et se promener avec un caniche susceptible.* Beaucoup de voleurs se tourneront vers des cibles ayant l'air plus riches et plus faciles, mais dans la nuit, un petit chien ne sera pas suffisant pour stopper un malfaiteur déterminé.

Mise en œuvre :

Camouflage : on peut cacher ou dévaluer les cibles du système et adopter comme politique de faire le moins de publicité possible sur les systèmes et leur contenu. On peut aussi configurer les modems pour ne décrocher qu'après un nombre de sonneries supérieur à ce que la plupart des programmes attendent, afficher des bannières de login génériques. "gtw" sera moins intrigant que "Global Thermonuclear War". Camoufler le système le rends moins utilisable et intuitif. Cela peut aussi rentrer en conflit avec les techniques de dissuasion qui mettent en valeur et exagèrent les défenses du système.

Mises en garde : elles informent les utilisateur que la sécurité du système est prise sérieusement en compte et exagèrent les pénalités encourues si une activité illégale est observée. Ces mises en garde sont également utiles du point de vue légal si par exemple on enregistre tout ce qui est tapé. Certains messages peuvent aussi être affichés lorsqu'un comportement intrusif est détecté. Cette méthode permet

cependant à l'intrus de savoir à quelles signatures ou quels comportements le système réagit.

Paranoïa : on essaye de donner l'impression à l'utilisateur qu'il est sur surveillance constante, que cela soit vrai ou non. La technique de la fausse alarme de voiture est le mécanisme le plus simple pour donner cette impression. Son pouvoir de dissuasion est cependant nul si l'on sait que c'est un faux. Pour contrer cela, on peut utiliser la technique de la caméra de surveillance, qui ne donne aucune indication sur le fait que quelqu'un surveille ou non. On sait qu'on peut l'être mais on ne sait pas à quel moment.

Obstacles : on essaye d'augmenter la quantité de temps et d'efforts qu'un intrus doit dépenser pour arriver à ses fins. Les obstacles, en particulier sur les passerelles, tentent de venir à bout de la patience de l'intrus, de gâcher son plaisir. On peut par exemple rallonger les temps d'exécution des commandes, afficher de fausses erreurs système, simuler une saturation des ressources pour l'exaspérer, sans toutefois lui donner l'impression d'être détecté. On peut aussi mettre en évidence des fichiers ou des comptes apparemment intéressants sur lesquels l'intrus passera du temps, révélera ses compétences mais qui ne lui apporteront rien. Toutes ces techniques peuvent cependant gêner les utilisateurs autorisés.

1.4 Détection

Les techniques de détection d'intrusion tentent de faire la différence entre une utilisation normale du système et une tentative d'intrusion et donnent l'alerte. Typiquement, les données d'audit du système sont parcourues à la recherche de signatures connues d'intrusion, de comportements anormaux ou d'autres choses intéressantes. La détection peut être faite en temps réel ou en tant qu'analyse post-mortem. Si la détection est en temps réel, le programme peut donner l'alerte, auquel cas le personnel qualifié pourra tenter de remédier à l'intrusion, en coupant le réseau ou en remontant la piste. Mais il peut probablement arriver après la fin de l'intrusion. Le programme peut également être couplé à un dispositif de contre-mesures pour pallier la lenteur humaine du personnel. Dans tous les cas, les techniques de détection d'intrusion seules n'empêcherons pas l'intrusion.

Analogie : *Avoir un sifflet et souffler dedans pour attirer l'attention de la police lorsqu'on est attaqué.* L'utilité est limitée si on est trop loin d'un magasin de donuts[2] ou si le syndrome des alarmes intempestives pousse les autorités à l'ignorer comme une fausse alarme. Il se peut également qu'on se rende compte du vol trop tard.

Mise en œuvre :
On se réfererra au chapitre 2.

[2] On ne niera pas l'origine américaine de l'exemple

1.5 Déflection

La déflection d'intrusion fait croire à un cyber-malfaiteur qu'il a réussi à accéder aux ressources système alors qu'il a été dirigé dans un environnement préparé et contrôlé. L'observation contrôlée à son insu d'un intrus qui dévoilerait alors tous ses tours est une excellente source d'information sans prendre de risques pour le vrai système. Certaines techniques de déflection peuvent être considérées comme une sorte de contre-mesure, mais le concept inclus aussi des techniques qui ne nécessitent pas l'accès au vrai système, comme les systèmes paratonnerre.

Analogie : *Transporter deux portefeuilles pour que, lors d'une attaque, le faux portefeuille avec la carte de crédit annulée se présente à l'intrus.* On peut ainsi apprendre comment le voleur opère, mais cela marche probablement seulement sur les voleurs débutants, et il peut être malcommode de se promener avec deux portefeuilles.

Mise en œuvre :

Faux systèmes en quarantaine : ils sont conçus pour faire croire aux intrus qu'ils sont sur le système cible. Cette déflection est faite par un frontal réseau tel qu'un routeur ou un firewall. Un faux système efficace encourage l'intrus à y rester assez longtemps pour que son identité et ses intentions soient découverts. Cependant, dédier une machine et des ressources pour l'entretenir coûte cher.

Faux comptes : ils sont conçus pour faire croire à l'intrus qu'ils utilisent un compte normal corrompu alors qu'ils sont bloqués dans un compte spécial aux accès limités. Dans ce cas, les contrôles de déflection sont inclus directement dans le système d'exploitation et les commandes du système. Cette technique élimine le besoin d'une machine séparée nécessaire à la mise en place d'un faux système mais doit compter sur la sécurité du système pour s'assurer d'une bonne isolation avec les ressources protégées. L'environnement ainsi construit devra aussi contenir assez de leurres pour confondre l'identité et les intentions de l'intrus. Cependant, maintenir un faux compte dont on ne peux sûrement pas s'échapper est difficile.

Systèmes et comptes paratonnerres : ils sont similaires aux faux systèmes et aux faux comptes comptes, mais au lieu de détourner l'intrus sur ces systèmes à son insu, on l'attire vers ces systèmes pour qu'il y aille de lui-même. Les paratonnerres[3] sont placés à proximité des systèmes à protéger, sont rendus attractifs, et sont équipés pour la détection d'intrusion. Ils n'ont rien a voir avec les ressources primaires qu'ils protègent et donc on n'a pas à se soucier de leurs performances ou capacités pour les utilisateurs autorisés. Leur installation et leur maintenance coûtent cependant cher, et leur efficacité repose sur le secret de leur existence.

1.6 Contre-mesures

Les contre-mesures donnent au système la capacité à réagir aux tentatives d'intrusions. Cette approche tente de remédier à la limitation des mécanismes qui reposent

[3] On emploie également le terme "pot de miel" (honeypot). Par exemple : http://project.honeynet.org/

sur l'attention continue de personnel humain. Il est en effet très difficile d'avoir du personnel dévoué 24h/24 à la réponse aux intrusions. De plus il ne pourra pas grand chose face à des attaques automatisées. Un système ayant été équipé pour le faire aura de plus grande chances de stopper l'intrusion. On court cependant le risque de mal réagir à un usage correct. On doit éviter le cas où un usager fait quelquechose d'inhabituel ou de suspect pour des raisons légitimes et se retrouve attaqué par un système de contre-mesures. De plus, il devient aisé pour un cyber-pirate de créer un déni de service pour personne donnée en usurpant son identité et en effectuant en son nom des opérations qui déclencheraient les contre-mesures.

Analogie : *Transporter une masse d'arme, fixer un piège à souris au portefeuille et apprendre le karaté pour contre-attaquer* On prend cependant le risque d'être poursuivi en justice pour avoir cassé le bras d'une inconnue qui vous offre des fleurs. Avec un portefeuille piégé, un voleur sera contré sans détection ni intervention de notre part. Mais même en tant qu'utilistateur autorisé du portefeuille, on peut se faire pincer les doigts si l'on oublie le piège ou si l'on manœuvre mal.

Mise en œuvre :

Les ICE (Intrusion Countermeasure Equipment) : les ICE [Gib84] permettent à un système de répondre en temps réel à une tentative d'intrusion, et ce de façon autonome. Les réactions suivantes peuvent être envisagées, par ordre de sévérité :

- Alerter, accroître l'aide au personnel
  - Signaler la détection
  - Accroître la collection d'audits sur l'utilisateur suspect, peut-être jusqu'au niveau des touches frappées
  - Alerter la personne chargée de la sécurité avec une alarme locale
  - Alerter la personne chargée de la sécurité même si elle est absente (nuit,..)
- Tenter de confirmer ou d'accroître les informations disponibles sur l'utilisateur
  - Ré-authentifier l'utilisateur ou le système distant pour remédier aux attaques profitant d'une session ouverte et oubliée ou de paquets forgés utilisant une connexion authentifiée
  - Alerter le personnel de sécurité pour avoir une confirmation visuelle ou vocale de l'identité et des intentions de la personne
- Minimiser les dommages potentiels
  - Journaliser les événements sur le système pour pouvoir revenir à un état considéré comme sûr
  - Ralentir le système ou rajouter des obstacles
  - Faire semblant d'exécuter les commandes (par exemple : mettre dans un tampon plutôt qu'effacer vraiment)
- Bloquer l'accès
  - Bloquer le compte local
  - Annuler les paquets concernés
  - Rechercher l'identité réseau et bloquer tous les comptes associés, faire le ménage dans les systèmes associés
  - Bloquer l'hôte totalement, le déconnecter du réseau
  - Déconnecter tout le système du réseau

Fichiers ou comptes piégés : ce sont les ressources nommées de façon séduisante

et placés stratégiquement qui leurrent les cyber-pirates et les amènent à révéler leurs activités. Une action est immédiatement déclenchée lors de l'accès à l'un de ces fichiers ou comptes. L'alarme peut être silencieuse et n'avertir que la personne chargée de la sécurité, ou peut déclencher des sanctions immédiates contre le cyber-criminel. Un candidat idéal comme compte piégé est le compte administrateur par défaut avec le mot de passe d'origine. Cette technique coûte peu en argent et en technique mais on doit faire attention à ce que des utilisateurs autorisés ne déclenchent pas l'alarme par mégarde.

Chapitre 2

Différentes approches pour la détection d'intrusion

On distingue deux grands types d'approches pour détecter des intrusions. La première consiste à rechercher des signatures connues d'attaques tandis que la seconde consiste à définir un comportement normal du système et à rechercher ce qui ne rentre pas dans ce comportement.

Un système de détection d'intrusions par recherche de signatures connaît ce qui est mal, alors qu'un système de détection d'intrusions par analyse de comportement connaît ce qui est bien.

On parle de détection de malveillances et de détection d'anomalies...

2.1 Détection de malveillances

La détection de malveillances fonctionne essentiellement par la recherche d'activités abusives, par comparaison avec des descriptions abstraites de ce qui est considéré comme malveillant. Cette approche tente de mettre en forme des règles qui décrivent les usages non désirés, en s'appuyant sur des intrusions passées ou des faiblesses théoriques connues. Les règles peuvent être faites pour reconnaître un seul événement qui est en lui-même dangereux pour le système ou une séquence d'événements représentant un scénario d'intrusion. L'efficacité de cette détection repose sur l'accuité et la couverture de tous les abus possibles par les règles.

Mise en œuvre :

Systèmes experts : ils peuvent être utilisés pour coder les signatures de malveillance avec des règles d'implication *si . . .alors*. Les signatures décrivent un aspect d'une attaque ou d'une classe d'attaque. Il est possible de rentrer de nouvelles règles pour détecter de nouvelles attaques. Les règles deviennent généralement très spécifiques au système cible et donc sont peu portables.

Raisonnement sur des modèles : on essaye de modéliser les malveillances à un niveau élevé et intuitif d'abstraction en termes de séquences d'événements qui

définissent l'intrusion. Cette technique peut être utile pour l'identification d'intrusions qui sont proches mais différentes. Elle permet aussi de cibler les données surs lesquelles une analyse approfondie doit être faite. Mais en tant qu'approche recherchant des signatures, elle ne peut que trouver des attaques déjà connues.

Analyse des transitions d'états : on crée un modèle tel que à l'état initial le système ne soit pas compromis. L'intrus accède au système. Il exécute une série d'actions qui provoquent des transitions sur les états du modèle, qui peuvent être des états ou l'on considère le système compromis. Cette approche de haut niveau peut reconnaître des variations d'attaques qui passeraient inaperçues avec des approches de plus bas niveau. USTAT (voir section 4.5) est une implémentation de cette technique.

Réseaux neuronaux : la flexibilité apportée par les réseaux neuronaux peut permettre d'analyser des données même si elles sont incomplètes ou déformées. Ils peuvent de plus permettre une analyse non-linéaire de ces données. Leur rapidité permet l'analyse d'importants flux d'audit en temps réel. On peut utiliser les réseaux neuronaux pour filtrer et sélectionner les informations suspectes pour permettre une analyse détaillée par un système expert. On peut aussi les utiliser directement pour la détection de malveillances. Mais leur apprentissage est extrêmement délicat, et il est difficile de savoir quand un réseau est prêt pour l'utilisation. On peut également lui reprocher son côté boite noire (on ne peut pas interpréter les coefficients).

Algorithmes génétiques : on définit chaque scénario d'attaque comme un ensemble pas forcément ordonné d'événements. Lorsqu'on veut tenir compte de tous les entremêlements possibles entre ces ensembles, l'explosion combinatoire qui en résulte interdit l'usage d'algorithmes de recherche traditionnels, et les algorithmes génétiques sont d'un grand secours.

La détection d'intrusion par recherche de scenarii repose sur une base de signatures d'intrusions et recherche ces signatures dans le journal d'audits.

On peut rapprocher les méthodes utilisées à celles que l'on peut rencontrer dans le domaine des antivirus, où on recherche la signature de certains programmes dans les fichiers du système informatique, ou encore dans le domaine de la génomique où l'on recherche une séquence d'ADN dans un brin, ou, plus généralement, tout ce qui s'apparente à l'appariement de séquences.

Inconvénients :

- Base de signatures difficile à construire.
- Pas de détection d'attaques non connues.

2.2 Détection d'anomalies

Cette approche se base sur l'hypothèse que l'exploitation d'une faille du système nécessite une utilisation anormale de ce système, et donc un comportement inhabituel de l'utilisateur. Elle cherche donc à répondre à la question « le comportement actuel de l'utilisateur ou du système est-il cohérent avec son comportement passé ? ».

Mise en œuvre :

Observation de seuils : on fixe le comportement normal d'un utilisateur par la donnée de seuils à certaines mesures (par exemple, le nombre maximum de mots de passe erronés). On a ainsi une définition claire et simple des comportements non acceptés. Il est cependant difficile de caractériser un comportement intrusif en termes de seuils, et on risque beaucoup de fausses alarmes ou beaucoup d'intrusions non détectées sur une population d'usagers non uniforme.

Profilage d'utilisateurs : on crée et on maintiens des profils individuels du travail des usagers, auxquels ils sont censés adhérer ensuite. Au fur et à mesure que l'utilisateur change ses activités, son profil de travail attendu se met a jour. Certains systèmes tentent de concilier l'utilisation de profils à court terme et de profils à long terme. Il reste cependant difficile de profiler un utilisateur irrégulier ou très dynamique. De plus, un utilisateur peut arriver à habituer lentement le système à un comportement intrusif.

Profilage de groupes : on place chaque utilisateur dans un groupe de travail qui montre une façon de travailler commune. Un profil de groupe est calculé en fonction de l'historique des activités du groupe entier. On vérifie que les individus du groupe travaillent de la manière que le groupe entier a défini par son profil. Cette méthode réduit drastiquement le nombre de profils à maintenir. De plus, un utilisateur peut beaucoup plus difficilement élargir le comportement accepté comme dans un profil individuel. Mais il est parfois dur de trouver le groupe le plus approprié à une personne : deux individus ayant le même poste peuvent avoir des habitudes de travail très différentes. Il est de plus parfois nécessaire de créer un groupe pour un seul individu.

Profilage d'utilisation de ressources : on observe l'utilisation de certains ressources comme les comptes, les applications, les mémoires de masse, la mémoire vive, les processeurs, les ports de communication sur de longues périodes, et on s'attend à ce qu'une utilisation normale n'induise pas de changement sur cette utilisation par rapport à ce qui a été observé par le passé. On peut aussi observer les changements dans l'utilisation des protocoles réseau, rechercher les ports qui voient leur trafic augmenter anormalement. Ces profils ne dépendant pas des utilisateurs et peuvent permettre la détection de plusieurs intrus qui collaboreraient. Les écarts par rapport au profil sont cependant très durs à interpréter.

Profilage de programmes exécutables : on observe l'utilisation des ressources du système par les programmes exécutables. Les virus, chevaux de Troie, vers, bombes logiques et autres programmes du même goût se voient démasqués en profilant la façon dont les objets du système comme les fichiers ou les imprimantes sont utilisés. Le profilage peut se faire par type d'exécutable. On peut par exemple détecter le fait qu'un daemon d'impression se mette à attendre des connexions sur des ports autres que celui qu'il utilise d'habitude.

Profilage statique c'est un profilage où la mise à jour du profil ne se fait pas en permanence mais seulement de temps en temps, et avec la bénédiction de la personne chargée de la sécurité. Dans le cas de profils utilisateurs, cela empêche l'usager d'élargir petit à petit son champ d'action. Cependant, les mises à jour peuvent être souvent nécessaires, et les fausses alarme peuvent rappeler l'histoire de « Pierre et le Loup ».

Profilage adaptatif : Le profil est mis à jour en permanence pour refléter les changements de comportement de l'utilisateur, du groupe ou du système. La personne en charge de la sécurité précise si la nouvelle activité est

- intrusive et des mesures doivent être prises,
- non intrusive, et peut être ajoutée au profil

– non intrusive, mais est une aberration dont la prochaine occurrence sera intéressante à connaître.

Profilage adaptatif à base de règles : cette technique diffère des autres profilages en ce que l'historique de l'utilisation est connu sous formes de règles. Contrairement à la détection de malveillances à base de règles, on n'a pas besoin des connaissances d'un expert. Les règles d'utilisation normale son générées automatiquement pendant la période d'apprentissage. Pour être efficace, beaucoup de règles sont nécessaires, et s'accompagnent d'autant de problèmes de performance.

Réseaux neuronaux : les réseaux neuronaux offrent une alternative à la maintenance d'un modèle de comportement normal d'un utilisateur. Ils peuvent offrir un modèle plus efficace et moins complexe que les moyennes et les déviations standard. L'utilisation de réseaux neuronaux doit encore faire ses preuves, et même s'ils peuvent s'avérer moins gourmands en ressources, une longue et minutieuse phase d'apprentissage est requise.

Approche immunologique : L'approche immunologique tente de calquer le comportement du système immunologique pour faire la différence entre ce qui est normal (le soi) et ce qui ne l'est pas (le non-soi). Le système immunologique montre en effet beaucoup d'aspects intéressants comme son mode d'opération distribué (il n'y a pas de système de contrôle central) qui lui permet de continuer à fonctionner même après des pertes, sa capacité à apprendre automatiquement de nouvelles attaques pour mieux réagir les prochaines fois qu'elles se présente, sa capacité à détecter des attaques inconnues, etc. On peut voir l'approche immunologique de la détection d'anomalies comme une méthode de détection d'anomalie où l'on utilise les techniques de détection des malveillance. En effet, les techniques de détection d'anomalie connaissent ce qui est bien et vérifient en permanence que l'activité du système est normale, alors que les techniques de détection de malveillance connaissent ce qui est mal et sont à sa recherche. L'approche immunologique propose de rechercher ce qui est mal en connaissant ce qui est bien. On devrais même dire que l'approche propose de rechercher ce qui n'est pas bien, et l'on peut se permettre la comparaison avec la notion de tiers-exclus en logique : on n'obtient pas ce qui est mal en prenant la négation de ce qui est bien. Cependant, les résultats obtenus sont satisfaisants.

Inconvénients :

– Choix délicat des différents paramètres du modèle statistique.
– Hypothèse d'une distribution normale des différentes mesures non prouvée.
– Choix des mesures à retenir pour un système cible donné délicat.
– Difficulté à dire si les observations faites pour un utilisateur particulier correspondent à des activités que l'on voudrait prohiber.
– Pour un utilisateur au comportement erratique, toute activité est normale.
– Pas de prise en compte des tentatives de collusion entre utilisateurs.
– En cas de profonde modification de l'environnement du système cible, déclenchement d'un flot ininterrompu d'alarmes (ex : guerre du Golfe).
– Utilisateur pouvant changer lentement de comportement dans le but d'habituer le système à un comportement intrusif.

2.3 Systèmes hybrides

Pour tenter de compenser quelques inconvénients de chacune des techniques, certains systèmes utilisent une combinaison de la détection d'anomalies et de la détection de malveillances. Par exemple, un compte d'administrateur aura un profil qui lui permet d'accéder à certains fichiers sensibles, mais il peut être utile de vérifier que des attaques connues ne sont pas utilisées contre ces fichiers. À l'inverse, utiliser des fichiers comportant le mot "nucléaire" ne caractérise aucune signature d'attaque, mais il peut être intéressant de savoir que cela est arrivé si ce n'était pas dans les habitudes de l'utilisateur.

Chapitre 3

Domaines impliqués dans la détection d'intrusions

3.1 Data mining

Le but est ici d'extraire des modèles descriptifs des énormes volumes de données d'audit. Plusieurs algorithmes du domaine du data mining peuvent être utiles. Parmi ceux qu'on peut trouver dans [LSM] et utilisés par [LS98], [LSM99] ou [LNY+00], ou peut noter :

Classification : la classification associe chaque élément de donnée à une plusieurs catégories prédéfinies. Ces algorithmes de classification génèrent des classifieurs, sous forme d'arbres de décision ou de règles. Une application dans la détection d'intrusion serait d'appliquer ces algorithmes à une quantité suffisante de données d'audit normales ou anormales pour générer un classifieur capable d'étiqueter comme appartenant à la catégorie normale ou anormale de nouvelles données d'audit.

Analyse de relations : on établit des relations entre différents champs d'éléments d'audit, comme par exemple la corrélation entre la commande et l'argument dans l'historique des commandes de l'interpréteur de commandes, pour construire des profils d'usage normal. Un programmeur, par exemple pourrait avoir une forte relation entre *emacs* et des fichier *C*. On définit ainsi des règles d'association.

Analyse de séquences : Ces algorithmes tentent de découvrir quelles séquences temporelles d'événements se produisent souvent en même temps.

On peut noter que [LNY+00] utilise le Common Intrusion Detection Framework (CIDF). Le CIDF est un effort pour développer des protocoles et des APIs pour permettre aux projets de recherche sur la détection d'intrusion de partager les informations et les ressources, et pour que les composants de détection d'intrusion puissent être réutilisés.

Un Internet Engineering Task Force (IETF) working group a été créé et nommé Intrusion Detection Working Group (IDWG).

Le CIDF est pour l'instant à l'état d'Internet Drafts qui sont en passe de devenir des RFC[1]

3.2 Agents

Plusieurs domaines de recherche dans les Mobile Agents Intrusions Detection System (MAIDS) sont ouverts par [JMKM99]. Certains sont mis en œuvre dans [BFFI+98], [HWHM98], [HWHM00] ou [JMKM00].

Détection multi-point La détection multi-point est faite en analysant les flux d'audit de plusieurs hôtes pour détecter des attaques distribuées ou autres stratégies d'attaques d'un réseau dans sa globalité. Il est rarement possible de transporter tous les flux d'audit à un IDS central, et même dans ce cas, la détection est difficile.

Les agents mobiles peuvent apporter le calcul distribué, le fait qu'on transporte l'analyseur au flux d'audit et non le flux d'audit à l'analyseur. Les agents pourraient détecter ces attaques, les corréler, et découvrir les stratégies d'attaques distribuées.

Architecture résistante aux attaques Une architecture hiérarchique est souvent utilisée pour des raisons de performances et de centralisation du contrôle. Cette conception a souvent plusieurs lignes de communication non-redondantes. Un intrus peut couper une branche de l'IDS, voire le désactiver totalement en le décapitant.

Les agents mobiles peuvent apporter quelques solutions à ce problème :
- une architecture complètement distribuée
- une architecture hiérarchique standard on un agent peut remplacer chaque nœud et ramener une fonctionnalité perdue
- des agents mobiles qui se déplacent quand une activité suspecte est détectée.

Partage de connaissances Souvent, plusieurs systèmes de détection d'intrusion différents fonctionnent sur un site. Idéalement, ils partageraient les informations sur les attaques récentes pour améliorer la détection d'attaques futures.

Même si ce n'est pas un domaine de recherche réservé aux MAIDS, ceux-ci permettent une approche plus naturelle de ce genre de choses.

Agents errants L'échantillonage aléatoire est utilisé avec succès depuis de longues années dans le domaine du contrôle de qualité, et les mathématiques sous-jacente sont bien comprises et les paramètres peuvent être calculés.

Chaque agent effectue un test spécifique et peut errer aléatoirement d'hôte en hôte. Quand des tests indiquent la possibilité d'une intrusion, des tests plus poussés peuvent être effectués sur l'hôte suspect.

[1] Request For Comment

Imprévisibilité Un intrus peut pénétrer dans un système sans être immédiatement détecté par l'IDS. Cela peut lui laisser le temps d'effacer ses traces ou de neutraliser l'IDS.

Un MAIDS reste vulnérable à cela mais se trouve néanmoins pourvu de quelques avantages. Lorsqu'un agent arrive sur un hôte, il transporte du code non encore altéré, et pourrait par exemple tester l'intégrité de la plateforme IDS locale. L'arrivée des agents et leur manière de fonctionner peuvent être imprévisibles, et il peut être très dur de rester inaperçu.

Diversité génétique Les IDS à base d'agents mobiles peuvent être vus comme un ensemble d'entités autonomes. Chaque agent peut être différent des autres par son code ou par les données sur lesquelles il travaille. Cependant, s'ils ne diffèrent que par leurs données, leurs tests peuvent devenir prévisibles.

Si chaque agent d'une même classe avait une façon différente de détecter la même chose, il serait autrement plus difficile de prévoir quoi que ce soit. Une manière de faire les choses serait de décrire ce qu'il faut détecter dans un langage standard et de laisser chaque instance de la classe trouver une manière de le détecter. Les agents avec un faible taux de détection pourraient essayer de muter en introduisant de légères modifications dans leur façon de détecter l'attaque.

3.3 Réseaux de neurones

Les réseaux neuronaux sont utilisés pour leur rapidité de traitement et leur relative résistance aux informations incomplètes ou déformées. [Can98] les utilise de deux manières différentes. Ils sont d'abord utilisés comme filtres pour filtrer et sélectionner les parties suspectes dans les données d'audit. Celles-ci sont ensuite analysées par un système expert. On peut ainsi réduire les fausses alarmes, et on peut même augmenter la sensibilité du système expert car il ne travaille que sur des données suspectes. Puis ils sont utilisés de façon à prendre seuls la décision de classer une séquence d'événements comme malveillante.

3.4 Immunologie

Le système immunitaire biologique est précisément conçu pour détecter et éliminer les infections. Il se montre capable de plusieurs propriétés que l'on aimerait voir figurer dans des systèmes artificiels.

Il est tout d'abord robuste, du fait de sa diversité, de sa distribution, de son dynamisme et de sa tolérance aux fautes. La diversité améliore la robustesse, deux éléments peuvent ne pas être vulnérables aux mêmes infections. Le fait que le système immunitaire soit distribué, que plusieurs composants interagissent localement pour obtenir une protection globale, permet de ne pas avoir de centre de contrôle, de point faible. Sa dynamique, le fait que ses agents sont continuellement créés, détruits et circulent augmentent la diversité temporelle et spatiale. Elle est tolérante aux fautes car l'effet d'un individu est faible, et quelques erreurs ne sont pas catastrophiques.

Ensuite, le système immunitaire est adaptatif, c'est-à-dire qu'il est capable d'apprendre et de reconnaître de nouvelles infections.

Enfin, il est autonome. Il ne nécessite pas de contrôle extérieur. De plus, comme il fait partie du corps, il se protège lui-même.

L'algorithme utilisé par le système immunitaire pour détecter les intrusions est appelé algorithme de sélection négative. Les lymphocytes sont appelés détecteurs négatifs parce qu'ils sont conçus pour se lier au non-soi, c'est-à-dire que lorsqu'un lymphocyte est activé, le système immunitaire répond à une intrusion.

Les lymphocytes sont créés avec des récepteurs générés aléatoirement. Le récepteur décide à quoi s'accrochera le lymphocyte. Comme ils sont générés aléatoirement, ils peuvent détecter aussi bien le soi que le non-soi. Aussi y a-t-il une période pendant laquelle le lymphocyte n'est pas mature et meurt s'il s'accroche à quelque chose. Les lymphocytes qui arrivent à maturité sont donc sensés détecter le non-soi. Lorsque le système immunitaire rencontre pour la première fois un certain type d'agents pathogènes, il produit une réponse primaire, qui peut prendre plusieurs semaines pour éliminer l'infection. Pendant cette réponse, il apprend à reconnaître ces agents et si une deuxième infection de ce type se déclare, il déclenchera une réponse secondaire en général assez efficace pour que l'infection passe inaperçue. La réponse primaire est lente parce qu'il peut n'y avoir qu'un petit nombre de lymphocytes qui s'attachent à eux. Pour accroître leur efficacité, chaque lymphocyte activé se clone, et on a ainsi une croissance exponentielle de la population qui peut détecter ces agents. Une fois l'infection éliminée, le système immunitaire garde cette population de lymphocytes qui avait une grande affinité avec ces agents pathogènes, alors que les autres lymphocytes ont une durée de vie de quelques jours. Les lymphocytes T sont créés et maturent uniquement dans le thymus. Bien que l'on rencontre en ce lieu la grande majorité des protéines du corps, il se peut que certains lymphocytes arrivent à maturité avec un détecteur qui s'attache à des cellules saines. Pour éviter cela, pour devenir actif, un lymphocyte nécessite une costimulation, c'est-à-dire que s'il s'attache sur une cellule, il lui faut aussi être stimulé par un second signal, qui est généralement un signal chimique généré quand le corps est agressé. Ce signal provient du système immunitaire lui-même ou d'autres cellules.

[FH] ou [FHS97] tentent d'appliquer vaguement quelques un de ces principes. Par contre, ARTIS (ARTificial Immune System) [HF00] calque parfaitement tous ces comportements, plus finement détaillés dans [Clo] ou [HF00], pour arriver à un résultat très encourageant.

3.5 Algorithmes génétiques

Comme expliqué dans [MA96], les algorithmes génétiques ont été proposés par John Holland dans les années 70 [Hol75]. Ils s'inspirent de l'évolution génétique des espèces, et plus précisément du principe de sélection naturelle. Il s'agit d'algorithmes de recherche d'optimums. On ne fait aucune hypothèse sur la fonction dont dont on recherche l'optimum. En particulier, elle n'a pas à être dérivable, ce qui est un avantage considérable sur toutes les méthodes classiques de recherche d'optimum. Un algorithme génétique manipule une population de taille constante d'individus représentés par une chaîne de caractères d'un alphabet. Chaque individu code chacun une solu-

tion potentielle au problème. La population est créée aléatoirement, puis elle évolue, génération par génération. À chaque génération, de nouvelles créatures sont créées en utilisant les plus fortes, tandis que les plus faibles sont éliminées, les adjectifs *faible* et *fort* étant bien sur déterminés par une fonction sélective, dépendant fortement de la fonction dont on cherche l'optimum. Des mutations peuvent aussi se produire, pour éviter à une population de perdre irrémédiablement une information sur la solution. On a donc trois transformations de la population : sélection, recombinaison et mutation.

Cela a inspiré [Mé96], article n'est pas resté sans suite puisque le logiciel GASSATA [Mé98] a ensuite été créé. (voir section 4.8).

Chapitre 4

Quelques programmes de détection d'intrusion existants

Pour une taxonomie assez complète et détaillée, on se référera à [Axe98], duquel son tirées quelques-unes des descriptions suivantes.

4.1 Haystack

Le prototype de Haystack [Sma88] a été développé pour la détection d'intrusions sur un système multi-utilisateur de l'Air Force. Il était conçu pour détecter six type d'intrusions

1. lorsqu'un utilisateur non-autorisé tente d'accéder au système
2. lorsqu'un utilisateur autorisé tente de prendre l'identité d'un autre
3. lorsqu'un utilisateur tente de modifier les paramètres relatif a la sécurité du système
4. lorsqu'un utilisateur tente d'extraire des données potentiellement sensibles du système
5. lorsqu'un utilisateur bloque l'accès aux ressources du système à d'autres utilisateurs
6. diverses attaques telles que l'effacement de fichier, etc.

Pour parvenir a ses fins, Haystack utilise les deux méthodes de détection : par détection d'anomalies et par signatures. La détection d'anomalies utilise un modèle par utilisateur décrivant le comportement de cet utilisateur dans le passé et un stéréotype qui spécifie le comportement générique acceptable pour cet utilisateur, évitant une dérive trop importante du premier modèle utilisateur. Il est ainsi impossible à un intrus d'habituer le système à un comportement intrusif.

4.2 MIDAS

MIDAS (Multics Intrusion Detection and Alerting System) [SSHW88] est construit autour du concept de détection d'intrusion heuristique. Les auteurs prennent exemple sur un administrateur humain en analysant comment il mènerais une analyse sur des journaux d'audit pour trouver des preuves d'intrusion. Il pourrait par exemple se dire que les intrusions se déroulent sans doute plus souvent tard dans la nuit quand le système est sans surveillance. Il pourrait faire l'hypothèse qu'un intrus, pour couvrir ses traces, utiliser plusieurs endroits d'où mener ses attaques. En combinant ces deux critères, on réduit fortement les événements à analyser.

MIDAS est un système expert à base de règles qui applique ce genre de raisonnements. Il utilise le Production Based Expert System Toolset (P-BEST[1]) et le peuple de trois catégories de règles :

Attaque immédiate : Les heuristiques d'attaque immédiate opèrent sans aucune connaissance de l'historique du système, sur une très petite fenêtre d'événements, typiquement un. Ces heuristiques sont statiques, elles ne changent qu'avec l'intervention de la personne chargée de la sécurité.

Anomalie d'un utilisateur : Les classes de règles pour les anomalies d'utilisateurs utilisent les profils statistiques des comportements passés des utilisateurs. Deux profils sont maintenus : le profil de session qui n'est valable que pendant la session courante et le profil utilisateur qui dure sur une longue période de temps. Le profil de session est mis à jour au login de l'utilisateur à partir de son profil utilisateur, qui lui est mis à jour à son tour à partir du profil de session au logout.

État du système Les heuristiques de l'état du système maintiennent des informations sur les statistiques du système en général, sans intérêt particulier pour les utilisateurs individuels, comme apr exemple le nombre total de login ratés par opposition au nombre de logins ratés d'un utilisateur particulier.

Les tests des auteurs ont montré que MIDAS était assez rapide pour l'analyse en temps réel, mais générais trop de fausses alarmes.

4.3 IDES

IDES (Intrusion Detection Expert System) [FJL+98, Lun90, LTG+92] repose sur l'hypothèse que le comportement d'un utilisateur reste à peu près le même au cours du temps, et que la manière dont il se comporte peut être résumée en calculant diverses statistiques sur son comportement.

IDES construit ses profils par groupes d'utilisateurs censés avoir un comportement proche et tente de corréler le comportement actuel d'un utilisateur avec son comportement passé et le comportement passé du groupe. Il observe trois types de sujets : les utilisateurs, les hôtes distants et les systèmes cible. AU total, 36 paramètres sont mesurés, 25 pour les utilisateurs, 6 pour les hôtes et 5 pour les systèmes cible. Toutes ses mesures font partie de ces deux catégories :

[1] P-BEST est un moteur de systèmes experts à chaînage avant. L'introduction de nouveaux faits dans sa base de faits déclenche la réévaluation de la base de règles, qui à son tour introduit de nouveaux faits. Le processus s'arrête lorsqu'aucune nouvelle règle n'est générée

Mesure catégorique : C'est une mesure de nature discrète et dont les valeurs appartiennent à un ensemble fini. On trouve par exemple les commandes invoquées par un utilisateur.

Mesure continue : C'est une fonction réelle. On a par exemple le nombre de lignes imprimées pendant la session ou la durée de la session.

IDES traite chaque enregistrement d'audit quand il arrive sur le système. Pour détecter des comportements anormaux pendant une session, alors que tous les paramètres de la session ne sont pas encore disponibles, IDES extrapole les valeurs et les compare au profil de l'utilisateur.

4.4 NIDES

NIDES (Next-generation Intrusion Detection Expert System) [AFV95] la continuation directe du projet IDES (voir section 4.3).

Plusieurs implémentations de NIDES existent. La version finale est hautement modulaire, avec des interfaces bien définies entre les composants. Il est centralisé dans le sens où l'analyseur tourne sur un hôte spécifique, et il collecte des données venant de divers hôtes à travers le réseau. La collecte d'audit se fait sur ces derniers en utilisant des sources d'audit variées.

Les composants clefs de NIDES sont :

Stockage persistant : ce composant propose les fonctions de gestion de l'archivage aux autres composants de NIDS.

Le composant *agend* : Il tourne sur tous les hôtes surveillés et se charge de lancer le convertisseur de données d'audit *agen* lorsque l'interface utilisateur de NIDS le lui demande. Il utilise le protocole RPC.

Le composant *agen* : il convertit les données d'audit au format canonique de NIDES. Les données converties sont ensuite données au système *arpool*.

Le composant *arpool* : ce composant collecte les données d'audit du composant *agen* et les transmet au composants d'analyse à base de règles ou d'analyse statistique lorsqu'ils le lui demandes. *arpool* tourne sur les hôtes surveillés.

Analyse statistique : ce composant calcule les données statistiques pour la détection d'intrusions. Il peut le faire en temps réel. Il transmet ses résultats au solveur.

Analyse à base de règles : ce composant recherche les signatures, en temps réel s'il le faut. Il transmet ses résultats au solveur.

Le solveur : il évalue et agit sur les données reçues des modules d'analyse statistique et à base de règles. Une seule action d'un utilisateur peut générer des dizaines d'alertes dans les composants inférieurs. Ce composant utilise toutes ces données pour prendre une décision. La personne en charge de la sécurité peut également lui demander d'ignorer les alertes relatives à certains objets.

L'archiveur : il a pour tâche d'archiver les données d'audit, les résultats d'analyses et les alertes.

L'analyseur hors-ligne : il permet de tester de nouvelles configurations sur de vielles données d'audit connues, en parallèle avec le système de détection d'intrusion en production.

L'interface utilisateur : elle est responsable de la communication avec la personne chargée de la sécurité. C'est elle qui permet de contrôler NIDES, et c'est elle qui reporte les alarmes. Une seule instance peut être lancée en même temps et elle doit tourner sur l'hôte qui fait tourner l'analyseur. L'implémentation utilise MOTIF sous X-Windows.

Les résultats montrent que NIDES est capable de détecter des anomalies avec des taux de fausse alertes et d'intrusions non détectées raisonnable.

4.5 USTAT

USTAT [Ilg91, Ilg93, IKP95] est une implémentation mature de l'analyse de transitions d'états pour la détection d'intrusions. Le système est initialement dans un état sûr et, à travers un certain nombre d'actions modélisées par des transitions d'états, peut se retrouver dans un état compromis.

Un exemple de scénario de pénétration présent dans les Unix BSD version 4.2 est :

`cp /bin/sh /usr/spool/mail/root`	on suppose que root n'a pas de mail
`chmod 4755 /usr/spool/mail/root`	on rend le fichier setuid
`echo | mail root`	on envoie un mail à root
`/usr/spool/mail/root`	on exécute le shell setuid root

la faille étant que `mail` ne change pas le bit setuid lorsqu'il change le propriétaire du fichier.

Ce scénario suppose certaines choses qui peuvent être vues comme un état de départ. Il n'est par exemple pas valide si root a du mail dans l'état courant du système. Chaque étape fait changer le système d'état, vers un état plus proche d'un état compromis. La première transition sera la création d'un fichier dans le répertoire de mail. La façon de le créer n'a pas d'importance, etc.

Le prototype est divisé en quatre modules :

Collecte des audits et pré-traitement : il collecte les données d'audit, les stocke pour études futures. Il les transforme également sous la forme canonique de USTAT : un triplet {*sujet*, *action*, *objet*}.

Base de connaissance : La base de connaissances contient la base de règles et la base de faits. La base de faits contient des informations sur les objets du système, par exemple les groupes de fichiers ou de répertoires qui partagent certaines caractéristiques qui les rendent vulnérables à certaines attaques. La base de règles contient les diagrammes de transition d'état qui décrivent un scénario particulier. Cela se compose d'une table de déscription des états, qui stocke les assertions sur chaque état, et la table de signatures d'actions.

Le moteur d'inférence : il évalue les nouveaux enregistrements d'audit, en utilisant les informations de la base de règle et de la base de faits, et mets a jour la base de faits avec des informations sir l'etat en cours. L'évaluation est semblable à un raisonnement par chaînage avant : les nouveaux faits, qui sont en fait les enregistrements d'audit, amènent à l'évaluation de toutes les règles qui pourraient dépendre sur les faits nouvellement arrivés, la base de faits est mise à jour pour en tenir compte, et une alerte est éventuellement déclenchée.

Le moteur de décisions : Il informe la personne chargée de la sécurité que le moteur d'inférences a détecté une possible intrusion. Dans le prototype, il alerte cette personne dès que l'une des transitions du scénario a été détectée et l'informe de préocédures à suivre pour empêcher les autres transitions. Les auteurs suggèrent d'ajouter à ce moteur des capacités à répondre aux intrusions.

4.6 IDIOT

IDIOT (Intrusion Detection In Our Time) [CDE+96] est un système de détection d'intrusions utilisant des réseaux de Pétri colorés pour la recherche de signatures. Par exemple l'attaque suivante (déjà citée dans la section 4.5) :

`cp /bin/sh /usr/spool/mail/root`	on suppose que root n'a pas de mail
`chmod 4755 /usr/spool/mail/root`	on rend le fichier setuid
`touch x`	on crée un fichier vide
`mail root < x`	on envoie un mail à root
`/usr/spool/mail/root`	on exécute le shell setuid root

peut être modélisée par le réseau de la figure 4.1.

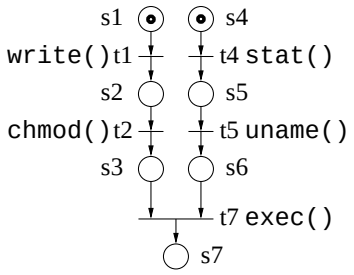

FIG. 4.1 – Une signature d'intrusion modélisée par un réseau de Pétri

Les différentes transitions sont conditionnées par les expressions boolélennes suivantes :

t_1 : *this.objet*=`/usr/spool/mail/root` et *FILE*←*this.object*.

t_2 : *this.objet=FILE*

t_4 : *FILE2*←*this.object*

t_5 : *FILE2=this.object*

t_7 : *this.prog*=`/usr/ucb/mail` et *this.args*=`root`

Une approche multi-couche est suggérée :

La couche informative : pour se débarasser des problèmes de compatibilité entre les données d'audit de différentes machines ou plateformes, cette couche permet aux couches supérieurs une interface commune avec les données.

La couche de signatures : elle décrit les signatures de comportements intrusifs, de façon indépendante de toute plateforme, en utilisant un modèle de machine virtuelle.

Le moteur d'appariements : il apparie les signatures du niveau précédent. Cela permet l'utilisation de n'importe quelle technologie d'appariement.

4.7 GrIDS

GrIDS [SCCC+96, CCD+99, SCCC+99] est un système de détection d'intrusions pour les gros réseaux utilisant des graphes. L'auteur propose une méthode pour construire des graphes de l'activité réseau. Les hôtes du réseau sont les nœuds du graphe et les connexions entre les hôtes sont les arêtes. La personne chargée de la sécurité donne un ensemble de règles qui permet de décider quel trafic entre les hôtes va représenter l'activité entre ces hôtes. Le graphe et les arêtes ont des attributs tels que les heures de connexion, etc., qui sont calculées par les règles données précédement.

La construction de l'activité du réseau repose sur le paradigme organisationnel d'une hierarchie de départements. Un département est formé de plusieurs hôtes, et il centralise les données d'audit et les combine pour générer le graphe du département, en se réferrant à l'ensemble de règles spécifié. Si des événements réseau d'un hôte du département impliquent un hôte hors du département, alors les graphes des deux départements peuvent être combinés, selon des règles spécifiées. Le nouveau graphe se compose donc de deux sommets, qui sont les départements et d'une arête qui spécifie le trafic entre ces deux départements. Ce processus est répété, et une hierarchie de départements est formée.

Les règles servent à plusieur choses. Elles permenttent de décider de combiner deux graphes en un graphe de niveau supérieur et comment le faire, comment calculer les attributs des graphes, les actions à prendre quand deux graphes sont combinés. Comme les règles peuvent être très compliquées, GrIDS utilise un langage qui permet de spécifier les politiques de comportement du réseau acceptables et inacceptables.

4.8 GASSATA

GASSATA [Mé98] (Genetic Algorithm for Simplified Security Audit Trail Analysis) est un outil de détection de malveillances. Il ne tient pas compte de la chronologie des événements pour pouvoir fonctionner dans un environnement distribué hétérogène où la construction d'un temps commun est impossible. L'algorithme est pesimiste dans le sens ou il tente d'expliquer les données d'audit par un ou plusieurs scenarii d'attaque. Mais c'est un problème de difficulté NP. Une méthode heuristique est donc utilisée : ce sont les algorithmes génétiques.

- Soit N_e le nombre de types d'événements
- Soit N_a le nombre d'attaques potentielles connues
- Soit AE une matrice $N_e \times N_a$ qui donne pour chaque attaque les événements qu'elle génère. $AE_{ij} \geq 0$ est le nombre d'événements de type i générés par l'attaque j.
- Soite R un vecteur de dimension N_e ou $R_i > 0$ est le poids associé à l'attaque i, poids proportionnel au risque inhérent du scénario d'attaque i.
- Soit O un vecteur de dimension N_e ou O_i est le nombre d'événements de type i dans le morceau d'audit analysé. O est le vecteur d'audit observé.

– Soit H un vecteur de dimension N_a ou $H_i = 1$ si l'attaque i est présente (d'après les hypothèses faites) et $H_i = 0$ sinon. H est un sous-ensemble des attaques possibles.

Pour expliquer le vecteur d'audit O par la présence d'une ou plusieurs attaques, on doit trouver le vecteur H qui maximise le produit $R \times H$ sous la contrainte $\forall i \in [1 \ldots N_a](AE \times H)_i \leq O_i$. C'est une approche pessimiste puisqu'on doit trouver le pire vecteur H.

Trouver le meilleur vecteur H est NP difficile. On ne peut donc pas appliquer les algorithmes classiques dans le cas où plusieurs centaines d'attaques sont recherchées, c'est-à-dire lorsque N_a est grand. L'heuristique employée est d'évaluer une hypothèse et de l'améliorer selon les résultats d'une évaluation. L'évaluation de l'hypothèse correspond à compter le nombre d'événements de chaque type générés par toutes les attaques de l'hypothèse. Si ces comptes sont plus petits ou egaux aux nombres d'événements enregistrés dans l'élément d'audit, alors l'hypothèse est réaliste. On doit alors la transformer en une hypothèse encore meilleure. L'algorithme utilisé pour obtenir une nouvelle hypothèse est un algorithme génétique.

Les algorithmes génétiques sont des algorithmes de recherche d'optimums basés sur la sélection naturelle dans une population. Une population est un ensemble de créatures artificielles. Ces créatures sont des chaînes de longueur fixe sur un alphabet donné, codant une solution potentielle du problème. La taille de la population est fixe. Elle est générée aléatoirement puis évolue. À chaque génération, de nouvelles créatures sont créées en utilisant tout ou parties des meilleures créatures de la population précédente. Le processus itératif de création d'une nouvelle population repose sur trois règles : la sélection des meilleurs individus, la reproduction ou croisement, qui permet l'exploration de nouvelles régions de l'espace des solutions) et la mutation, qui protège la population contre les pertes irrecupérables d'information. L'évolution de la population se fait jusqu'à ce qu'elle remplisse une certaine condition, qui est usuellement le fait qu'un individu est solution du problème.

Dans le cas de GASSATA, un individu sera un vecteur H possible, et la fonction à maximiser sera une fonction du produit à maximiser $R \times H$ tenant compte du réalisme des hypothèses.

4.9 Hyperview

Hyperview [DBS92] est un système avec deux composants principaux. Le premier est un système expert qui observe les données d'audit à la recherche de signes d'intrusions connus, l'autre est un réseau de neurones qui apprend le comportement d'un utilisateur et déclenche une alerte lorsque les données d'audit dévient du comportement appris.

Les données d'audit peuvent venir d'une multitude d'endroits à différents niveaux de détail, comme par exemple s'interesser à toutes les touches tapées par l'utilisateur, ou simplement aux différentes commandes passées. Les auteurs notent que plus les données sont détaillées et brutes, plus les chances de détecter correctement une intrusion sont élevées. Plus également les temps de calcul et la taille des données a stocker seront importants.

Pour utiliser un réseau de neurones, l'hypothèse a été prise que les données d'audit

seraient des séries temporelles et que l'utilisateur génèrerait une série ordonnée d'événements choisis parmi un ensemble fini.

Les recherches se sont portées au début sur un réseau de neuronne prennant sur ses N entrées une fenêtre temporelle de taille N sur la série d'événements d'audit. Cependant, certains problèmes résultent de cette approche : N est statique. Si sa valeur devait changer, il faudrait réentrainer complètement le réseau. Si N n'etait pas correctement choisi, les performances du système serait beaucoup réduites. Pendand les périodes ou les événements sont quasi-stationnaires, une grande valeur de N est préférable, alors que pendant les phases de transition, une petite valeur serait plus adaptée.

Les auteurs conclurent alors que les corrélations entre les motifs d'entrée ne seraient pas pris en compte car ces types de réseaux n'apprendraient à reconnaître que des motifs fixes dans l'entrée et rien de plus. Ils choisirent alors d'utiliser des réseaux récurrents, (des réseaux avec une architecture de Gent [GS92]) c'est-à-dire qu'une partie de la sortie était connectée à l'entrée. Cela crée une mémoire interne au réseau. Les événements sont entrés un par un dans le réseau, qui dispose maintenant d'une mémoire à court terme : les valeurs d'activation des neurones grace à la boucle de la sortie vers l'entrée, et d'une mémoire à long terme : les coefficients des neurones du réseau.

Le réseau de neurone a autant de sorties que le nombre d'éléments différents d'événements dans les données d'audit. Après entrainement, le réseau doit être en mesure de prédire le prochain événement. Si le niveau d'une des sorties est très supérieur à tous les autres, le réseau prédit un certain événement avec confiance. Si tous les niveaux de sortie sont faibles et dans les mêmes valeurs, cela signifie soit que la série n'a jamais été rencontrée, soit qu'il y a trop de possibilités. Si plusieurs sorties se détachent à niveau égal, chaque candidat est probable.

Le réseau de neurones a été connecté à deux systèmes experts. Le premier supervisait l'entrainement du réseau, l'empêchant par exemple d'apprendre des comportements anormaux. Le deuxième recherchais des signatures d'intrusion connues dans les données d'audit. Le réseau de neurones peut être vu comme un filtre pour le système expert. Les deux systèmes experts mettent ensuite leurs résultats en commun pour décider ensuite de déclencher ou non une alarme.

Les résultats furent prometteurs.

Deuxième partie

EIDF

Chapitre 5

L'architecture EIDF

Les techniques de détection d'intrusions sont multiples et il reste difficile de s'en faire un idée précise tant qu'on ne les a pas manipulées.

Chaque outil utilise son propre moteur et ses propres sources d'audit. Il est difficile de comparer plusieurs modèles sur le même terrain, ou même de comparer plusieurs terrains sous le même modèle.

Il pourrait être intéressant de disposer de plusieurs modèles interchangeables et de pouvoir choisir sa source d'information indépendemment du modèle choisi.

5.1 Architecture

5.1.1 Aperçu

Dans cet esprit, on voit donc deux objets bien séparés : la source d'audit et l'analyseur. La source d'audit auscultera la ou les parties du système produisant les données d'audit qu'elle désire. Ces données peuvent être de première ou seconde main. Elles seront converties en une suite d'objets d'un certain type, représentant une suite d'événements. La source d'audit mettra à disposition ces objets à la demande, à un analyseur qui les acceptera un par un. Cette architecture on ne peut plus simple permet, au prix de quelques restrictions, de mettre en relation n'importe quelle source d'audit avec n'importe quel analyseur. Cependant, pour faire fonctionner tout cela, ces deux objets doivent être supportés par une charpente, un programme qui se chargera de construire ces objets et de les mettre en relation.

On peut voir un schéma de cette architecture sur la figure 5.1.

5.1.2 Les source d'audit

Le but de la source d'audit est de mettre à disposition des événements provenant d'une partie du système observée pour l'occasion. Ces événements peuvent être de n'importe quel type : nombre entier, réel, chaîne de caractère, *n*-uplet, objet plus com-

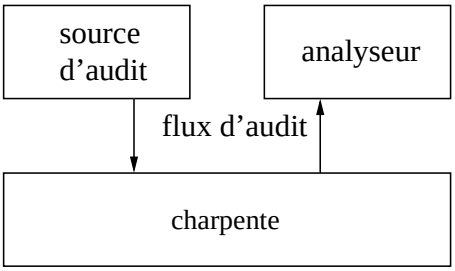

FIG. 5.1 – Aperçu de l'architecture EIDF

plexe. La suite de ces événements se veut une description de la vie de la partie du système observée. Cette description peut être plus ou moins détaillée, mais elle doit permettre de se faire une idée du comportement de cette partie du système.

5.1.3 Les analyseurs

Un analyseur ne se fait aucune idée *a priori* du type ou de la structure de l'objet. Il est seulement capable de savoir si deux objets sont égaux ou non. S'il est souhaitable de travailler sur des classes d'objets, c'est à la source d'audit de prendre l'ensemble quotient. Cela exclus dans l'analyseur la notion de proximité. C'est le prix à payer pour avoir une indépendance vis-à-vis de la source[1].

Un analyseur dispose d'une base de connaissances persistante. La plupart des analyseurs doivent d'abord remplir leur base avant de pouvoir détecter quoi que ce soit. Ce remplissage se fait durant une phase d'apprentissage, où l'analyseur reçoit le flux d'audit. Ensuite, en mode vérification, il peut utiliser ses connaissances pour décider si un événement est normal ou non.

De même qu'on ne peut pas savoir si un chien se gratte à partir d'une image d'un flux vidéo, les analyseurs peuvent difficilement se faire une idée du comportement du système en en traitant que le dernier événement du flux. La plupart des analyseurs utiliseront donc une fenêtre remontant plus ou moins loin dans le passé du flux. À chaque nouvel événement, on a donc, pour une fenêtre de taille k, un k-uplet d'événements, et le but est de savoir si ce k-uplet est normal ou anormal. Il est donc très important dans ce cas là que le flux soit élémentaire.

Un analyseur peut fournir certaines statistiques sur le flux traité, comme le nombre d'événements rencontrés. Il peut également mettre à disposition son estimation de la couverture de l'espace des séquences d'événements utilisées par la source d'audit.

5.1.4 La charpente

La principale fonction de la charpente est de mettre en relation la source d'audit et l'analyseur. Elle se charge également de les instancier et d'ouvrir et fermer la source

[1] On pourra bien sur enfreindre cette loi et faire des suppositions *a priori* sur la structure des événements, mais cet analyseur ne fonctionnera pas avec toutes les sources

d'audit. On peut voir cela dans l'algorithme simplifié 5.1. Enfin, c'est elle qui traitera les paramètres et configurera l'analyseur et la source en fonction.

Alg. 5.1 – Grands traits du fonctionnement de la charpente

```
Instancier et configurer l'analyseur
Instancier et configurer la source d'audit
Ouvrir la source d'audit
while estimation de couverture de l'analyseur < limite fixée do
    Demander un événement à la source d'audit
    if vérification then
        Fournir l'événement à l'analyseur pour vérification
    else
        Demander à l'analyseur d'apprendre l'événement
    end if
end while
Fermer de la source d'audit
```

5.2 Multiplexage

5.2.1 Motivations

Nous avons vu que les analyseurs essayent souvent de se faire une idée du comportement du système à partir de l'enchaînement de plusieurs événements. Cela nécessite une source qui observe la même chose tout le temps. Un analyseur ne pourra pas se faire une idée de la dynamique de la partie du système qu'il étudie si on lui présente quelques images d'un plan sur le chien, puis quelques autres d'un plan sur la maison. Il n'est pas capable d'interpréter les événements, et encore moins de détecter les changements de plans. Par contre la source le peut.

De même qu'il n'y a pas beaucoup de films en un seul plan, beaucoup de sources sont un enchevêtrement de différentes lignes d'action. Par exemple, chaque connexion réseau est analysable, mais il faut les séparer et les analyser chacune séparément des autres. Observer les appels systèmes effectués par un seul processus est intéressant mais limité : il est plus intéressant d'être capable d'observer plusieurs voire tous les processus. Mais on doit analyser chaque flux d'appels systèmes séparément, alors qu'ils sont multiplexés temporellement.

5.2.2 Mise en œuvre

L'architecture vue précédemment semble peu adaptée à supporter un flux multiplexé. Pourtant, il suffit en fait d'un super-analyseur qui démultiplexe le flux en flux

élémentaires et instancie autant d'analyseurs que de flux élémentaires pour résoudre le problème. Pour pouvoir démultiplexer le flux, la source doit l'étiqueter. Le super-analyseur s'attend donc à une certaine structure d'événement. C'est le seul analyseur du modèle pour lequel une telle chose est tolérée. Pour quand même conserver l'interchangeabilité, il acceptera également les flux non multiplexés et les transmettra tels quels à une instance d'analyseur. De même, un analyseur recevant un flux multiplexé traitera chaque événement étiqueté comme un événement, et verra donc deux mêmes événements étiquetés différemment comme différents. Bien qu'il s'accommodera très bien d'un flux multiplexé, celui-ci n'a pas beaucoup de sens. La chronologie entre des événements étiquetés différemment ne doit pas être prise en compte, et le sera pourtant par la plupart des analyseurs.

Dans certains cas, par exemple si on trace les appels systèmes de tous les processus du système, c'est à la source d'audit d'aller chercher les événements des différents flux élémentaires. Il est alors facile de les étiqueter et de les multiplexer. Dans d'autres, comme par exemple si on capture les paquets du réseau, les flux élémentaires sont mélangés, et c'est à la source d'audit d'interpréter chaque événement pour pouvoir l'étiqueter.

L'étiquette d'un événement se compose tout d'abord d'un identificateur de flux, attribué par la source d'audit. Il est unique et permet d'associer un flux élémentaire à une instance d'analyseur. Cet identificateur sera un objet quelconque acceptant l'opérateur d'égalité. Cela peut être le PID du processus tracé, si on trace les appels systèmes de plusieurs processus. Ou encore le quadruplet *(IP1, port1, IP2, port2)* identifiant une connexion TCP. L'étiquette se compose ensuite d'un type de flux, attribué également par la source d'audit, qui décidera que deux flux élémentaires doivent avoir le même comportement. Par exemple, un processus `sendmail` devra avoir le même comportement que n'importe quel autre processus `sendmail`. De même qu'une connexion TCP sur le port 80 devra toujours avoir le même comportement que n'importe quelle autre connexion sur le port 80. On peut noter que l'identificateur de flux ne signifie rien et peut changer d'une session sur l'autre sans problème, tandis que l'identificateur de type de flux est un identificateur persistant. Les traces d'un processus `sendmail` doivent être identifiées de façon identique d'une session sur l'autre, par exemple en utilisant la chaîne de caractères `"sendmail"`.

5.2.3 Le Super-analyseur

Le fonctionnement du super-analyseur est assez simple. Tout d'abord, on lui fournit comme paramètre un modèle d'analyseur à utiliser. Il est ensuite prêt à recevoir des événements.

Si l'événement qu'il reçoit ne provient pas d'un flux multiplexé, c'est-à-dire si ce n'est pas un triplet composé d'un identificateur de flux, d'un identificateur de type de flux et d'un événement, les identificateurs prennent une valeur par défaut, ce qui revient à traiter le flux comme un flux multiplexé contenant un unique flux élémentaire.

Si l'événement provient d'un flux multiplexé, il contient son identificateur de flux et son identificateur de type de flux. Chaque flux élémentaire est associé à sa propre instance d'analyseur grâce à son identificateur de flux. À chaque nouvel identificateur de flux, un nouvel analyseur est instancié et lui est associé.

Par contre, toutes les instances d'analyseurs d'une même classe de flux partagent la même base de connaissance. Pour cela, une instance de l'analyseur, que l'on appellera instance reine, est spécialement créée. Son rôle est uniquement de contenir la base de connaissances, cette dernière ne pouvant exister hors de l'objet qui l'a créée.

Pour chaque nouveau type de flux rencontré, deux cas se présentent. Si l'on est en mode vérification, l'événement est considéré comme anormal ou est ignoré selon la configuration du super-analyseur. Dans le cas où l'on est en mode apprentissage, un nouvel analyseur est instancié. L'instance créée est associée à ce type de flux, et devient une instance reine.

Pour chaque nouveau flux élémentaire, on instanciera un analyseur dont on fera pointer la base de connaissance sur la base de connaissance de l'instance reine associée au type du flux élémentaire.

La base de connaissances du super-analyseur est donc en fait l'ensemble des instances reines, contenant chacune leur propre base de connaissances, ainsi que le dictionnaire les reliant avec chaque type de flux que le super-analyseur connaît.

On peut voir sur la figure 5.2 la façon dont le super-analyseur traite le flux multiplexé. Le flux est d'abord démultiplexé selon son identificateur de flux, puis passé en entrée pour apprentissage ou vérification à une instance d'analyseur *a*. Pour chaque type de flux élémentaire déjà rencontré, il existe une instance reine *A* de l'analyseur, dont le rôle est uniquement de créer et de garder une base de connaissance *k* utilisée par toutes les instances d'analyseurs traitant les flux de ce type. L'ensemble de ces instances reines ainsi que la façon de les associer à un type de flux constitue la base de connaissances *K* du super-analyseur.

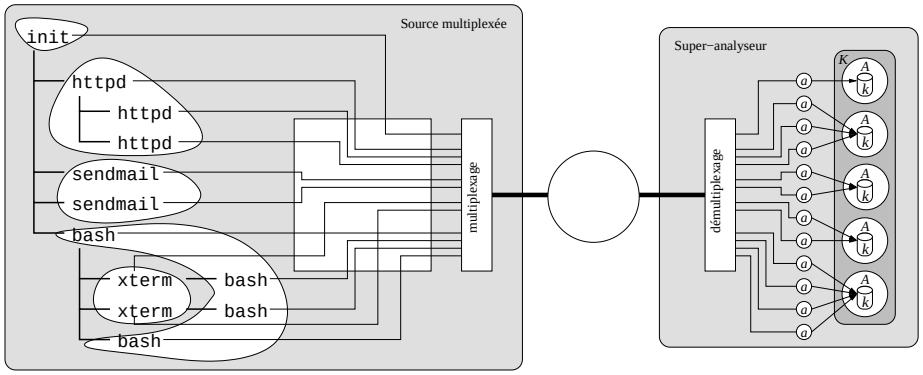

FIG. 5.2 – Fonctionnement du super-analyseur

Faiblesses : Ce modèle ne permet pas de faire passer des informations telles que la fin d'un flux élémentaire, comme la mort d'un processus tracé, ou la fin d'une connexion TCP. Les conséquences directes sont que la mémoire occupée par une instance d'analyseur n'est jamais libérée et que certains identificateurs comme le PID d'un processus ou le quadruplet *(IP1, port1, IP2, port2)* peuvent être réutilisés.

Le deuxième problème n'en est pas vraiment un si la source d'audit respecte les spécifications précédemment énoncées, c'est-à-dire si chaque identificateur de flux est

réellement unique. On peut programmer la source d'audit de manière à rendre chaque identificateur réellement unique, quitte à numéroter les occurrences des identificateurs identiques.

Une solution au premier problème peut être de donner un âge à chaque instance d'analyseur. Cet âge sera remit à zéro à chaque utilisation et incrémenté de temps en temps, jusqu'à un âge limite ou l'instance sera détruite. C'est une sorte d'algorithme ramasse-miettes imparfait, car on n'a pas l'assurance que ce qui est détruit ne servait vraiment plus. Cependant, une destruction par erreur n'a pas forcément un impact énorme. Cette méthode est toutefois difficile à régler correctement.

Un autre problème peut surgir de l'accès concurrent de plusieurs analyseurs à la même base. Le passage du flux par la charpente impose une sérialisation de ce flux. On est donc certain que deux analyseurs ne peuvent pas accéder à la base en même temps. Cependant, dans certains cas la base peut être influencée par l'ordre dans lequel les événements arrivent. Par exemple, dans le cas d'un réseau de neurones, la base de connaissances est ce même réseau de neurones. Il va donc être confronté de façon entremêlée à des événements de flux élémentaires différents, comme si le super-analyseur ne les démultiplexait pas, ou plus exactement ne démultiplexait que selon les identificateurs de types de flux et non selon les identificateurs de flux. Dans le cas de réseaux non cycliques, cela ne pose pas ou peu de problèmes puisque l'ordre d'apprentissage des événements a une influence qui n'était de toute façon pas maîtrisée. Mais dans le cas de réseaux cycliques, ou donc il existe une mémoire à court terme des événements passés, cela peut avoir de graves répercussions.

5.3 Détails d'implémentation du modèle

Toute l'implémentation a été faite en Python[2], un langage de très haut niveau, interprété, et orienté objet. La programmation orientée objet permet d'arriver au niveau de modularité souhaité.

5.3.1 Charpente

La charpente se charge tout d'abord de parcourir les paramètres de la ligne de commande. Beaucoup d'options peuvent lui être fournies. Sa syntaxe est la suivante :

```
eidf.py {-l|-c} [-a] [-h] [-d|D dumpfile] [-t threshold] [-v] [-u]
        [-m model] [-M 'model-parameters']  [-f knowledge-file]
        [-s audit-source] [-S 'source-parameters']
```

- `{-l|-c}` choisis le mode apprentissage (`-l`) ou vérification (`-c`)
- `[-m model]` choisis le modèle à utiliser. Par défaut, le premier modèle trouvé sera utilisé
- `[-M 'model-parameters']` précise les paramètres à fournir à l'analyseur. Chaque analyseur précise les paramètres dont il a besoin.
- `[-f knowledge-file]` précise le fichier dans lequel sera lue et sauvée la base de connaissance de l'analyseur. Par défaut, il s'agit du fichier `profile.eidf`.
- `[-s audit-source]` permet de choisir la source d'audit utilisée.

[2] http://www.python.org

– `[-S 'source-parameters']` permet de fournir les paramètres à passer à la source d'audit
– `[-a]` en mode apprentissage, demande à l'analyseur d'ajouter ce qu'il va apprendre à la base de connaissances plutôt que de l'écraser.
– `[-d|D dumpfile]` demande d'écrire tous les paramètres que l'analyseur peut fournir dans un fichier, événement par événement. Si l'option est en majuscule, les données sont ajoutées au fichier au lieu de l'écraser.
– `[-t threshold]` précise une limite en pourcentage comme condition d'arrêt dans le cas de l'apprentissage. Le programme s'arrête lorsque l'estimation de couverture dépasse la limite fournie.
– `[-u]` demande à la charpente de retirer les étiquettes d'un flux multiplexé. Cela permet d'utiliser directement un analyseur sans passer par le super-analyseur dans le cas particulier ou la source ne multiplexe qu'un flux élémentaire, mais lui attribue des identificateurs de flux différents d'une session sur l'autre.
– `[-v]` demande à la charpente d'afficher des informations sur les opérations en cours. L'option peut être passée plusieurs fois pour afficher encore plus d'informations.
– `[-h]` affiche la syntaxe ainsi qu'une liste des différents analyseurs et des différentes sources d'audit disponibles.

Le code complet de cette partie de l'architecture se trouve en annexe dans la section C.1.

5.3.2 Sources d'audit

Chaque source d'audit est implémentée sous forme d'un objet. Toutes les sources d'audit ont une interface identique, elles héritent d'une classe mère. On peut ainsi remplir le critère d'interchangeabilité.

Les différentes méthodes de cet objet sont, en plus de son constructeur :
– Ouverture du flux : `open()`
– Fermeture du flux : `close()`
– Obtention d'un événement : `getevent()`

De plus chaque classe doit avoir deux attributs. L'attribut `name` est le nom de la source et permet de faire référence à cette source en tant que paramètre dans la ligne de commande de la charpente. L'attribut `parameters` permet à la charpente, lorsqu'elle annonce la disponibilité d'une source d'audit, de préciser les paramètres que cette dernière peut accepter. Certains paramètres peuvent être obligatoires.

On peut voir la classe mère dans le listing 5.1. Elle se contente de renvoyer comme événement chaque ligne lue sur l'entrée standard.

Listing 5.1 – Classe mère des sources d'audit

```
class Audit_source:
    name="Template"
    parameters=""
    def __init__(self,options=[]):
        if options:
            self.parse_options(options)
    def parse_options(self, options):
        pass
```

```
    def open(self):
        pass
    def getevent(self):
        return sys.stdin.readline()
    def close(self):
        pass
```

La méthode `getevent()` peut renvoyer n'importe quel type d'objet, puisque les analyseurs ne font aucune supposition *a priori* sur le type d'événements qu'ils reçoivent. Cependant, dans le cas d'une source multiplexée, cette méthode doit renvoyer une liste composée d'un identificateur de flux, d'un identificateur de type de flux et de l'événement. Chacun de ces trois objets peut être de n'importe quel type. Encore une fois, seul l'opérateur d'égalité est utilisé.

La méthode `parse_options()` ne fait pas partie à proprement parler de l'interface. Elle existe surtout pour séparer le processus d'analyse des options du reste des opérations du constructeur `__init__()`.

5.3.3 Analyseurs

Chaque analyseur est également implémenté sous forme d'objet. Un analyseur est un objet beaucoup plus complexe qu'une source d'audit. Ils ont cependant en commun les attributs `name` et `parameters`, ainsi que la méthode `parse_options()`, extraite du constructeur.

Les différentes méthodes de cet objet sont :
- Chargement de la base de connaissances à partir d'un fichier : `load_knowledge(file)`
- Sauvegarde de la base de connaissances dans un fichier : `save_knowledge(file)`
- Retour d'une référence sur la base de connaissances : `get_knowledge()`
- Changement de la base de connaissance à partir d'une référence : `set_knowledge(knowledge)`
- Apprentissage d'un événement : `learn(event)`
- Vérification de la normalité d'un événement : `normal(event)`
- Retour de statistiques sous forme brute pour journaliser dans un fichier : `logs()`
- Retour de statistiques mises en forme : `stats()`
- Retour d'une estimation de la couverture des différents types d'événements parcourus : `completed()`

L'estimation de couverture n'est qu'un calcul simple sur les différentes statistiques maintenues par la méthode d'apprentissage `learn()`. Elle compte le nombre d'événements qu'elle a reçu et la position du dernier événement qui lui a appris quelquechose. Cette dernière variable peut être plus ou moins dure à évaluer. S'il s'agit d'enregistrer tous les événements, un événement pas encore rencontré suffit à décider de mettre à jour cette variable avec la position du compteur sans aucun problème. Dans le cas de réseaux neuronaux, la décision est plus complexe. À partir de ces deux variables n et d, respectivement le nombre d'événements analysés et la position du dernier événement digne d'intérêt, la couverture estimée peut être calculée avec la formule $c = \frac{n-d}{n}$. Ainsi, si on ne rencontre rien de nouveau pendant une longue période, l'estimation va croître. Si on finit par rencontrer quelquechose de nouveau, par exemple à la position n_1, alors l'estimation va repartir à 0. Il devient raisonnable de penser que le minimum est d'attendre encore au moins $2n_1$ pour quelquechose d'encore nouveau. L'estimation atteint

alors 50% à ce moment.

La base de connaissance d'un analyseur est un objet dont le type et la structure sont laissées à la discrétion de l'analyseur.

La classe mère (listing 5.2) peut être utilisée comme analyseur, mais elle n'apprend rien, elle a une connaissance nulle et considère tout comme normal.

Listing 5.2 – Classe mère des analyseurs

```
class Analyzer:
    name="Template"
    parameters=""
    def __init__(self,options=[]):
        if options:
            self.parse_options(options)
        self.knowledge=None
        self.count=1
        self.learned=0
        self.lastlearned=1
    def parse_options(self,options):
        pass
    def load_knowledge(self,file):
        f=open(file,"rb")
        try:
            data=pickle.load(f)
            if type(data) != type(()): raise ValueError
            if len(data) != 2: raise ValueError
            if type(data[0]) != type(""): raise ValueError
            f.close()
        except ValueError:
            raise ValueError,"%s is not a EIDF knowledge file" % file
        sig,self.knowledge=data
        if sig != self.name:
            raise ValueError,"signature \"%s\" expected. Found \"%s\" in %s"
                            % (self.name,sig,file)
    def save_knowledge(self,file):
        f=open(file,"wb")
        pickle.dump((self.name,self.knowledge),f)
        f.close()
    def get_knowledge(self):
        return self.knowledge
    def set_knowledge(self,knowledge):
        self.knowledge=knowledge
    def completed(self):
        return 1.0*(self.count-self.lastlearned)/self.count
    def stats(self):
        return ["Completed: %3.2f%%" % (100*self.completed()),
                "Events: %5i" % self.count,
                "Learned: %5i" % self.learned,
                "Learning slope: %3.4f" % (self.learned*1.0/self.count)]
    def logs(self):
        return "%i %f %i %i"
               % (self.count,self.completed(),self.learned,self.lastlearned)
    def learn(self,event):
        self.count=self.count+1
```

```
    def normal(self,event):
        self.count=self.count+1
        return 1
```

Chapitre 6

Sources d'audit

6.1 Lecteur d'enregistrements

Le lecteur d'enregistrements est le complémentaire de l'analyseur enregistreur, dont le rôle est d'apprendre tout l'enchaînement des événements (voir section 7.1). Le lecteur est donc capable de lire la base de connaissances de l'enregistreur et de permettre de rejouer plusieurs fois le même scénario enregistré pour tester plusieurs analyseurs ou pour tester plusieurs paramètres pour le même analyseur.

Listing 6.1 – Classe de lecture d'enregistrements

```
class Player(Audit_source):
    name="Player"
    parameters="-f file"
    recorder="Recorder"
    def __init__(self, file="",options=[]):
        self.file=file
        Audit_source.__init__(self,options)

    def open(self):
        f=open(self.file,"rb")
        try:
            data=pickle.load(f)
            if type(data) != type(()): raise ValueError
            if len(data) != 2: raise ValueError
            if type(data[0]) != type(""): raise ValueError
            f.close()
        except ValueError:
            raise ValueError,"%s is not a EIDF knowledge file" % file
        sig,self.data=data
        if sig != self.recorder:
            raise ValueError,"signature \"%s\" expected. Found \"%s\" in %s"
                        % (self.recorder,sig,file)
        if not self.file:
            raise ValueError, "no file specified"
        try:
            self.f=open(self.file)
```

```
        except IOError,msg:
            raise ValueError, msg

    def parse_options(self,options):
        try:
            opts=getopt.getopt(options,"f:")
        except getopt.error,errmsg:
            raise getopt.error,"%s source: %s" %(self.name,errmsg)
        if len(opts[1]) > 0:
            raise getopt.error,"Parameters unrocognized : %s etc." % opts[1][0]
        for opt,parm in opts[0]:
            if opt == "-f":
                self.file=parm
        if not self.file:
            raise getopt.error,"%s source: -f option is mandatory" % self.name

    def getevent(self):
        if not self.data:
            raise EOFError,"No more audit data"
        a=self.data[0]
        del(self.data[0])
        return a

    def close(self):
        self.f.close()
```

6.2 Appels système d'un processus

L'utilisation de cette source consiste à espionner tous les appels système effectués par un processus bien choisi du système en utilisant les méchanismes proposés par le système d'exploitation.

C'est la source d'audit utilisée par [FHSL96].

L'ensemble des enchaînements des appels systèmes est fixé à la compilation du programme et de ses bibliothèques. On a donc de bonnes raisons de penser que c'est un excellent indicateur du comportement d'un processus. Pour continuer à aller dans ce sens, on peut regarder un graphe de l'enchaînement des appels système d'un processus, `sendmail` dans le cas de la figure 6.1, et l'on s'apercevra que les motifs sont plutôt réguliers.

Pour tracer un processus, certains noyaux dont le noyau Linux mettent à disposition un appel système `ptrace()`. Cet appel système fournit à un processus un moyen d'observer et de contrôler l'exécution d'un autre processus. Il est en particulier utilisé par les programmes de débogage pour placer des points d'arrêt.

Cette source d'audit prend le PID d'un processus comme paramètre. C'est un paramètre obligatoire. Lors de l'ouverture du flux d'audit, le processus exécutant l'objet va s'attacher au processus à tracer. L'exécution de ce dernier est alors stoppée à chaque appel système. Lorsqu'un événement est demandé, l'analyseur attend que le processus s'arrête à un appel système s'il n'est pas déjà arrêté, récupère le numéro de l'appel sys-

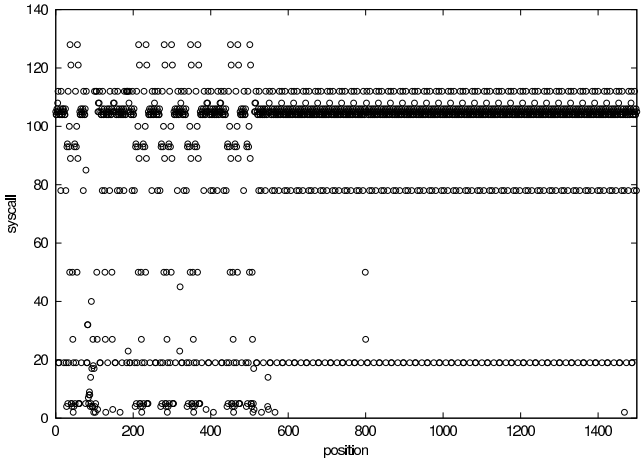

FIG. 6.1 – Enchaînement d'appels systèmes de `sendmail`

tème, et relance le processus tracé jusqu'à ce qu'il effectue un nouvel appel système. Pendant ce temps, la source retourne le numéro de l'appel système.

On peut voir le code de l'objet sur le listing 6.2.

Listing 6.2 – Classe de traçage d'appels systèmes

```
class Stracepid(Audit_source):
    name="stracePID"
    parameters="-p pid"
    def __init__(self, pid=0,options=[]):
        self.pid=pid
        Audit_source.__init__(self,options)

    def parse_options(self,options):
        try:
            opts=getopt.getopt(options,"p:")
        except getopt.error,errmsg:
            raise getopt.error,"%s source: %s" %(self.name,errmsg)

        if len(opts[1]) > 0:
            raise getopt.error,"Parameters unrocognized : %s etc." % opts[1][0]
        for opt,parm in opts[0]:
            if opt == "-p":
                try:
                    self.pid=int(parm)
                except ValueError,str:
                    raise getopt.error,"pid : %s" % str
        if self.pid == 0:
            raise getopt.error,"%s source: no pid specified" % self.name

    def open(self):
        if self.pid == 0:
            raise ValueError, "no pid specified"
        try:
            ptrace.attach(self.pid)
```

```
        except OSError,msg:
            raise ValueError, msg

        ptrace.syscall(self.pid,0)

    def getevent(self):
        print os.waitpid(self.pid,0)
        try:
            syscall=ptrace.peekuser(self.pid,ORIG_EAX)
        except OSError:
            raise EOFError,"No more audit data"

        ptrace.syscall(self.pid,0)
        return syscall;

    def close(self):
        ptrace.detach(self.pid,0)
```

Cette implémentation est limitée par le fait qu'elle ne trace que le processus désigné et ne se soucie pas de ses fils.

6.3 Lecteur des données de [FHSL96]

Afin de comparer certains résultats avec ceux de [FHSL96], et aussi afin de pouvoir utiliser ces données d'audit assez largement adoptées, une source d'audit a été créée. Les données fournies sont des traces d'appels système de processus `sendmail`, dans le cas d'utilisation normale, mais également lors de différentes attaques. La source est multiplexée car plusieurs processus `sendmail` sont tracés en même temps.

On peut voir le code de cette classe dans le listing 6.3. On peut noter que l'identificateur de type de classe ne varie jamais et valeur utilisée, la chaîne de caractères `"sendmail"`, est codée en dur.

Listing 6.3 – Classe de lecture des données de [FHSL96]

```
class ForrestDisk(Audit_source):
    name="ForrestDisk"
    parameters="-f file"
    def __init__(self, file="", options=[]):
        self.file=file
        Audit_source.__init__(self,options)

    def parse_options(self,options):
        try:
            opts=getopt.getopt(options,"f:")
        except getopt.error,errmsg:
            raise getopt.error,"%s source: %s" %(self.name,errmsg)

        if len(opts[1]) > 0:
            raise getopt.error,"Parameters unrocognized : %s etc." % opts[1][0]
        for opt,parm in opts[0]:
            if opt == "-f":
```

```
            self.file=parm
        if not self.file:
            raise getopt.error,"%s source: -f option is mandatory" % self.name

    def open(self):
        if not self.file:
            raise ValueError, "no file specified"
        try:
            self.f=open(self.file)
        except IOError,msg:
            raise ValueError, msg

    def getevent(self):
        a=string.split(self.f.readline())
        if not a:
            raise EOFError,"No more audit data"
        try:
            try:
                return (int(a[0]),"sendmail",int(a[1]))
            except IndexError:
                raise ValueError
        except ValueError:
            raise ValueError,"%s source: format error in %s" % (self.name,self.file)

    def close(self):
        self.f.close()
```

Chapitre 7

Les analyseurs

7.1 Enregistreur

Cet analyseur sert à enregistrer tout l'enchaînement des événements provenant d'une source. Sa base de connaissance est en fait la liste ordonnée de tous les événements rencontrés. C'est cette base que la source d'audit de restitution lira (voir section 6.1).

La classe (listing 7.1) est très simple et profite au maximum de son héritage. Lorsqu'elle est appelée pour vérification, elle comparera chaque événement fourni avec celui occupant la même position dans sa base de connaissances.

La base de connaissance pourrait encore être améliorée en stockant également le temps écoulé entre chaque événement, de manière à être en mesure, si cela est demandé, de reproduire la séquence de manière parfaite.

Listing 7.1 – Classe de l'enregistreur

```
class Recorder(Analyzer):
    name="Recorder"
    parameters=""
    def __init__(self,options=[]):
        Analyzer.__init__(self,options)
        self.knowledge=[]

    def learn(self,event):
        self.count=self.count+1
        self.knowledge.append(event)

    def normal(self,event):
        self.count=self.count+1
        return ((self.count-2 < len(self.knowledge)) and
                (self.knowledge[self.count-2]==event))
```

7.2 Analyse par correspondance exacte

Cet analyseur travaille sur une fenêtre de taille *k* paramétrable. Il a besoin d'une phase d'apprentissage, durant laquelle il mémorise tous les *k*-uplets d'événements. Sa base de connaissance est un dictionnaire indexé par le premier événement d'une fenêtre. L'élément référencé est encore un dictionnaire indexé cette fois par le deuxième élément de la fenêtre. Il y a autant de niveaux d'imbrications que de d'éléments dans la fenêtre. La structure obtenue s'apparente alors à un arbre ayant *k* niveaux.

Ensuite, pendant la phase de vérification, il considère comme anormal tout *k*-uplet qui n'est pas présent dans la base, i.e. qui n'a encore jamais été rencontré.

On peut voir le code de cette classe dans le listing 7.2.

Listing 7.2 – Classe de l'analyseur par correspondance exacte

```
class ExactMatch(Analyzer):
    name="ExactMatch"
    parameters="[-w learn_window_size]"
    def __init__(self,learn_window_size=4,options=[]):
        self.learn_window_size=learn_window_size
        Analyzer.__init__(self,options)
        self.knowledge={}
        self.learn_window=[]
        self.check_window=[]
        self.failed_window=[]

    def parse_options(self,options):
        try:
            opts=getopt.getopt(options,"w:")
        except getopt.error,errmsg:
            raise getopt.error,"%s model: %s" %(self.name,errmsg)
        if len(opts[1]) > 0:
            raise getopt.error,"%s model: Parameters unrocognized : %s etc."
                                %(self.name,opts[1][0])
        for opt,parm in opts[0]:
            if opt == "-w":
                try:
                    self.learn_window_size=int(parm)
                except ValueError,str:
                    raise getopt.error,"learn window size : %s" % str

    def learn(self,event):
        self.count=self.count+1
        w=self.learn_window
        w.append(event)
        if len(w) > self.learn_window_size:
            del(w[0])

        flag=0
        node=self.knowledge
        for ev in w:
            if not node.has_key(ev):
                node[ev]={}
                flag=1
```

```
            node=node[ev]

        if flag:
            self.learned=self.learned+1
            self.lastlearned=self.count

    def normal(self,event):
        self.count=self.count+1
        w=self.check_window
        w.append(event)
        if len(w) > self.learn_window_size:
            del(w[0])

        try:
            node=self.knowledge
            for ev in w:
                node=node[ev]
            return 1
        except KeyError:
            return 0
```

7.3 Analyse par correspondance avec seuil

La phase d'apprentissage est identique à l'analyse par correspondance exacte : toutes les occurrences de k-uplets sont mémorisées.

Pour la phase de vérification, une fenêtre de taille l avec $l \geq k$ est utilisée. On compte le nombre de k-uplets parmi les $l-k+1$ de la fenêtre de longueur l qui ne sont pas dans la base, et cette fenêtre est considérée comme anormale si ce nombre excède un seuil s.

On peut voir le code de cette classe dans le listing 7.3. On peut remarquer que cette classe n'hérite pas directement de la classe `Analyzer` comme la plupart des autres classes mais de la classe `ExactMatch` (section 7.2) dont une partie du comportement est semblable.

Listing 7.3 – Classe de l'analyseur par correspondance avec seuil

```
class Threshold(ExactMatch):
    name="Threshold"
    parameters="[-w learn_window_size] [-W check_window_size] [-t threshold]"
    def __init__(self,learn_window_size=4, check_window_size=20,
                 alert_threshold=4, options=[]):
        self.check_window_size=check_window_size
        self.alert_threshold=alert_threshold
        ExactMatch.__init__(self,learn_window_size,options=options)

    def parse_options(self,options):
        try:
            opts=getopt.getopt(options,"w:W:t:")
        except getopt.error,errmsg:
            raise getopt.error,"%s model: %s" %(self.name,errmsg)
```

```
        if len(opts[1]) > 0:
            raise getopt.error,"%s model: Parameters unrocognized : %s etc."
                                % (self.name,opts[1][0])
        for opt,parm in opts[0]:
            if opt == "-w":
                try:
                    self.learn_window_size=int(parm)
                except ValueError,str:
                    raise getopt.error,"learn window size : %s" % str
            elif opt == "-W":
                try:
                    self.check_window_size=int(parm)
                except ValueError,str:
                    raise getopt.error,"check window size : %s" % str
            elif opt == "-t":
                try:
                    self.alert_threshold=int(parm)
                except ValueError,str:
                    raise getopt.error,"threshold : %s" % str

    def normal(self,event):
        self.count=self.count+1
        f=self.failed_window
        w=self.check_window
        w.append(event)
        if len(w) > self.learn_window_size:
            del(w[0])

        try:
            node=self.knowledge
            for ev in w:
                node=node[ev]
        except KeyError:
            f.append(1)
        else:
            f.append(0)

        if len(f) > self.check_window_size:
            del(f[0])

        return reduce(lambda x,y:x+y, f) < self.alert_threshold
```

7.4 Modèle de “Forrest”

L’analyseur utilisé dans [FHSL96] mémorise, dans sa phase d’apprentissage, pour chaque événement, les $k-1$ événements qui le suivent usuellement et la place à laquelle ils le suivent.

Par exemple, pour la séquence d’événements *ABAACDBAC* avec une fenêtre de taille $k=4$, on mémorisera que *A* est suivi d’abord de *A*, *B* ou *C*, puis de *A*, *C* ou *D*, et enfin de *A*, *B* ou *D*. *B* pour sa part est tout le temps suivi de *A*, puis de *A* ou *C*, puis de

C. *C* est tout le temps suivi de *D* puis de *B* puis de *A*. *D* enfin est suivi de *B* puis de *A* puis de *C*.

Lors de la phase de vérification, on utilise une fenêtre de taille $l \geq k$. Pour chacun des l événements de cette fenêtre, on regarde si l'événement qui le suit à la 1, 2, ..., $k$ième position figurait bien à cette place durant l'apprentissage. On compte chaque erreur, et on divise par le nombre maximal d'erreurs possibles M, qui vaut

$$M = (k-1)(l-k+1) + (k-2) + \ldots + 1 = (k-1)\left(l - \frac{k}{2}\right)$$

Si on utilise la base générée par l'apprentissage de l'exemple précédent et la fenêtre de taille 7 *ABCDABC*, on obtient :

Le premier k-uplet ($k = 4$) est *ABCD*. On a vu que *A* doit être suivi de *A*, *B* ou *C*. Il est dans ce cas suivi de *B*, donc pas d'erreur. On devrait ensuite trouver *A*, *C* ou *D*. On trouve *C*, donc pas d'erreur. On devrait enfin trouver *A*, *B* ou *D*. On trouve *D* donc toujours pas d'erreur. On passe au k-uplet suivant : *BCDA*. *B* doit être suivi de *A*. Or on trouve *C*, donc une erreur. Puis *A* ou *C*, or on trouve *D*, donc une autre erreur. Enfin de *C*, mais on trouve *A* donc une autre erreur.

On peut voir l'implémentation de ce modèle dans le listing 7.4.

Listing 7.4 – Classe de l'analyseur de "Forrest"

```
class Forrest(Analyzer):
    name="Forrest"
    parameters="[-w window_size] [-W check_window_size] [-t threshold]"
    def __init__(self,window_size=4,check_window_size=8,threshold=10,options=[]):
        self.window_size=window_size
        self.check_window_size=check_window_size
        self.threshold=threshold
        Analyzer.__init__(self,options)

        self.knowledge={}
        self.learn_window=[]
        self.check_window=[]
        self.failed_window=[]
        self.max_error=(self.window_size-1)*
                       (self.check_window_size-self.window_size*0.5)/100.0

    def parse_options(self,options):
        try:
            opts=getopt.getopt(options,"w:W:t:")
        except getopt.error,errmsg:
            raise getopt.error,"%s model: %s" %(self.name,errmsg)

        if len(opts[1]) > 0:
            raise getopt.error,"%s model: Parameters unrocognized : %s etc."
                               %(self.name,opts[1][0])
        for opt,parm in opts[0]:
            if opt == "-w":
                try:
                    self.window_size=int(parm)
                except ValueError,str:
```

```
                raise getopt.error,"window size : %s" % str
            elif opt == "-W":
                try:
                    self.check_window_size=int(parm)
                except ValueError, str:
                    raise getopt.error,"check window size : %s" % str
            elif opt == "-t":
                try:
                    self.threshold=int(parm)
                except ValueError, str:
                    raise getopt.error,"threshold : %s" % str

    def learn(self,event):
        self.count=self.count+1
        w=self.learn_window

        w.append(event)
        if len(w) > self.window_size:
            del(w[0])

        flag=0
        k=self.knowledge;
        if not k.has_key(w[0]):
            k[w[0]]=[{}]*(self.window_size-1)
            flag=1

        for ev in range(len(w)-1):
            if not  k[w[0]][ev].has_key(w[ev]):
                k[w[0]][ev][w[ev]]=None
                self.learned=self.learned+1
                flag=1
        if flag:
            self.lastlearned=self.count

    def normal(self,event):
        self.count=self.count+1
        f=self.failed_window
        w=self.check_window
        ws=self.window_size
        cws=self.check_window_size

        w.append(event)
        if len(w) > cws:
            del(w[0])
        if len(f) >= cws:
            del(f[0])
        f.append(0)
        if len(w) < cws:
            return 1

        for e in range(ws-2):
            evt=w[cws-ws+e]
            try:
                self.knowledge[evt][ws-2-e].has_key(w[-1])
```

```
            except KeyError:
                f[cws-ws+e]=f[cws-ws+e]+1
        return reduce(lambda x,y:x+y, f) < self.threshold*self.max_error
```

7.5 Réseaux neuronaux

Lorsqu'on parle apprentissage et reconnaissance, on ne peut s'empêcher de penser réseaux neuronaux. Nous allons donc voir ce que peut permettre l'utilisation de réseaux neuronaux dans notre cas.

On pourra se référer à l'annexe A pour plus de détails. Afin d'implémenter ce type d'analyseurs, quelques classes permettant de créer et d'utiliser des réseaux neuronaux de type perceptron multi-couche ont été développées. On peut les trouver en annexe à la section A.4

Pour l'apprentissage de séries temporelles par un réseau neuronal, on utilise en général une fenêtre temporelle de la taille de l'entrée du réseau, fenêtre qu'on fournit en entrée du réseau, comme on peut le voir sur la figure 7.1.

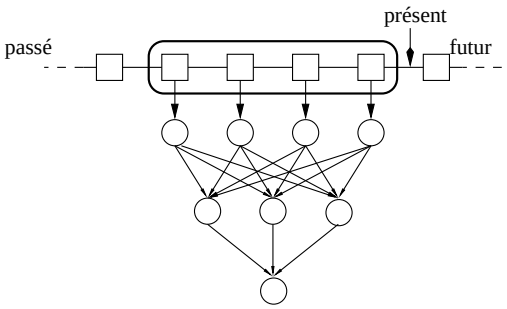

FIG. 7.1 – Utilisation d'un réseau neuronal pour une série temporelle

Le premier problème qui se pose est que chaque événement est un objet de type inconnu. Cela ne pose néanmoins pas de problème de les convertir en nombre, le premier événement reçu portant le numéro 1, et chaque événement pas encore référencé recevant le numéro libre suivant.

Le deuxième problème est que cette façon de numéroter ne donne pas de sens à la notion de proximité. Les événements numérotes 1 et 3 peuvent être beaucoup plus semblables que chacun avec le numéro 2. Comme l'analyseur ne peut pas interpréter les événements, il n'y a aucun moyen d'assigner des numéros avec du sens.

Un moyen de résoudre ce problème est d'assigner un neurone d'entrée à chaque type d'événements. Il sera le seul excité lorsque cet événement sera reçu. On peut voir cela sur la figure 7.2

On essayera aussi, sans trop de convictions, de combiner tous les événements de la fenêtre dans le même vecteur d'entrée, en pondérant chaque événement en fonction de son ancienneté, comme on peut le voir sur la figure 7.3.

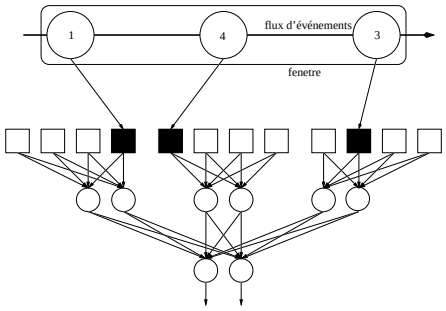

FIG. 7.2 – Utilisation de plusieurs entrées pour chaque événement

Il y a également plusieurs façons de faire dire à un réseau de neurones ce qu'il pense d'un *k*-uplet d'événements. La méthode la plus simple est d'avoir un seul neurone de sortie qui renvoie 1 lorsqu'il pense que tout est normal et 0 sinon. On peut améliorer légèrement la chose en rajoutant un neurone de sortie qui lui renvoie 0 si tout va bien et 1 sinon. On peut voir cette méthode illustrée dans la figure 7.2

On peut également utiliser le réseau pour prédire le prochain événement en fonction des événements passés, comme on peut le voir sur la figure 7.3. Il y a autant de neurones que de types d'événements. Si un des neurones a une activation très supérieure à tous les autres, c'est que le réseau donne une prédiction avec confiance. Si au contraire tous les neurones de sortie ont une activation faible et environ du même niveau, c'est que le réseau est incertain sur ce qui va suivre. Si enfin plusieurs neurones de sortie on une activation importante, c'est qu'il y a plusieurs candidats.

Comme les analyseurs utilisant des réseaux de neurones ont une structure commune, une classe a été créée pour qu'ils puissent tous hériter de cette spécialisation de la classe `Analyzer`. On peut voir son code dans le listing 7.5. Trois méthodes ont été ajoutées, qui sont en rapport avec les réseaux neuronaux.

- `build_net()` construit le réseau neuronal
- `build_input()` construit, à partir de la fenêtre des événements passés et de l'événement tout juste reçu, l'entrée qui va être donnée au réseau.
- `build_desired()` construit, à partir de l'événement tout juste reçu la sortie qu'on désire obtenir, soit pour apprentissage, soit pour comparaison avec la prédiction.

Listing 7.5 – Classe modèle des analyseurs à base de réseaux neuronaux

```
class NeuralAnalyzer(Analyzer):
    name="NeuralAnalyzer"
    parameters="[-w window-size] [-m max_event] [-t threshold]"
    def __init__(self,window_size=4, max_event=50,threshold=0.1,options=[]):
        self.window_size=window_size
        self.max_event=max_event
        self.threshold=threshold
        Analyzer.__init__(self)

        self.rej={}
        self.rejnb=0
        self.learn_window=[]
        self.check_window=[]
```

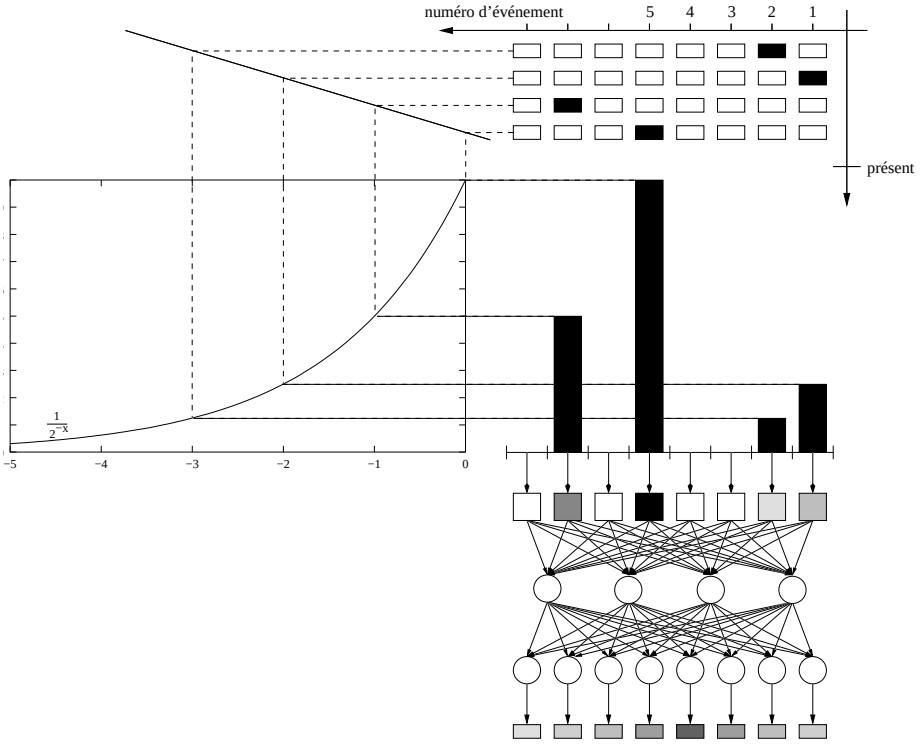

FIG. 7.3 – Prédiction d'un événement : qui viendra après 2, 1, 7 et 5 ?

```
        self.knowledge=[{},self.build_net()]

    def build_net(self):
        m=self.max_event
        c1=[1/sqrt(m)]*m
        return NeuralNet([[Neuron(Sigmoid(),c1),Neuron(Sigmoid(),c1)]])

    def build_input(self, win, event):
        input=[0]*self.max_event
        input[event]=1
        for i in range(len(win)):
            input[win[i]]=min(1,input[win[i]]+1.0/(2.0**(i+1)))
        return input

    def build_desired(self, event):
        return [1,0]

    def parse_options(self,options):
        try:
            opts=getopt.getopt(options,"w:m:t:")
        except getopt.error,errmsg:
            raise getopt.error,"%s model: %s" %(self.name,errmsg)
```

```
        if len(opts[1]) > 0:
            raise getopt.error,"%s model: Parameters unrocognized : %s etc."
                                % (self.name,opts[1][0])
        for opt,parm in opts[0]:
            if opt == "-w":
                try:
                    self.window_size=int(parm)
                except ValueError,str:
                    raise getopt.error,"window size : %s" % str
            elif opt == "-W":
                try:
                    self.max_event=int(parm)
                except ValueError,str:
                    raise getopt.error,"maximum event number : %s" % str
            elif opt == "-t":
                try:
                    self.threshold=float(parm)
                except ValueError,str:
                    raise getopt.error,"threshold : %s" % str

    def learn(self,event):
        self.count=self.count+1
        w=self.learn_window

        d,n=self.knowledge
        if not d.has_key(event):
            num=len(d)
            if num >= self.max_event:
                if not self.rej.has_key(event):
                    self.rej[event]=1
                else:
                    self.rej[event]=self.rej[event]+1
                self.rejnb=self.rejnb+1
                return
            d[event]=num

        n.learn(self.build_input(w,d[event]),self.build_desired(d[event]))

        w.insert(0,d[event])
        if len(w) > self.window_size:
            del(w[-1])

    def normal(self,event):
        self.count=self.count+1
        w=self.check_window
        d,n=self.knowledge

        if not d.has_key(event):
            return 0

        out=array(n.calc(self.build_input(w,d[event])))
        desired=array(self.build_desired(d[event]))
        print out
        w.insert(0,d[event])
```

```
        if len(w) > self.window_size:
            del(w[-1])

        return sqrt(sum((desired-out)**2)) < self.threshold

    def stats(self):
        s=Analyzer.stats(self)
        s.append("Rejected (too many events) : %i events of %i different types"
                 % (self.rejnb,len(self.rej)))
        return s
```

On a ainsi implémenté deux analyseurs à base de réseaux de neurones, qu'on peut voir sur les listings 7.6 et 7.7. Le premier correspond au réseau représenté sur la figure 7.3. Le deuxième donne une prédiction du en sortie, comme le premier, mais prend une entrée du type de celle représentée sur la figure 7.2. Dans les deux cas, le nombre de neurones de la couche médiane a été choisi au hasard. Les coefficients sont initialisés à $\frac{1}{\sqrt{n}}$ où n est le nombre d'entrées du neurone considéré, comme conseillé dans [Sím98].

Listing 7.6 – Classe d'un analyseurs à base de réseaux neuronaux

```
class Neural(NeuralAnalyzer):
    name="Neural"
    def __init__(self,window_size=4, max_event=50,threshold=0.1,options=[]):
        NeuralAnalyzer.__init__(self,window_size,max_event,threshold,options)

    def build_net(self):
        m=self.max_event
        c1=[1/sqrt(m)]*m
        f1=Sigmoid()
        layer1=[]
        for a in range(m/2):
            layer1.append(Neuron(f1,c1,learn_rate=0.5))
        c2=[1/sqrt(m/2)]*(m/2)
        f2=Sigmoid()
        layer2=[]
        for a in range(m):
            layer2.append(Neuron(f2,c2,learn_rate=0.5))
        return NeuralNet([layer1,layer2])

    def build_input(self, win, event):
        input=[0]*self.max_event
        for i in range(len(win)):
            input[win[i]]=min(1,input[win[i]]+1.0/(2.0**i))
        return input

    def build_desired(self, event):
        desired=[0]*self.max_event
        desired[d[event]]=1
        return desired


    def normal(self,event):
        self.count=self.count+1
        w=self.check_window
```

```
        d,n=self.knowledge

        if not d.has_key(event):
            return 0

        input=[0]*self.max_event
        for i in range(len(w)):
            input[w[i]]=1.0/(2.0**i)
        desired=[0]*self.max_event
        desired[d[event]]=1

        out=array(n.calc(input))
        v=out[d[event]]
        out=clip(out-v,0,1)

        w.insert(0,d[event])
        if len(w) > self.window_size:
            del(w[-1])

        return max(out) < 0.1 and sum(out) < 1
```

Listing 7.7 – Classe d'un analyseurs à base de réseaux neuronaux

```
class Neural2(NeuralAnalyzer):
    name="Neural2"
    def __init__(self,window_size=4, max_event=50,threshold=0.1,options=[]):
        NeuralAnalyzer.__init__(self,window_size,max_event,threshold,options)

    def build_net(self):
        m=self.max_event
        c1=[1/sqrt(m)]*m
        c1=c1*self.window_size
        f1=Sigmoid()
        layer1=[]
        for a in range(2*m):
            layer1.append(Neuron(f1,c1,learn_rate=0.5))
        c2=[1/sqrt(m)]*(2*m)
        f2=Sigmoid()
        layer2=[]
        for a in range(m):
            layer2.append(Neuron(f2,c2,learn_rate=0.5))
        return NeuralNet([layer1,layer2])

    def build_input(self, win, event):
        input=[]
        win=win+[0]*(self.window_size-len(win))
        for evt in win:
            inp=[0]*self.max_event
            inp[evt]=1
            input=input+inp
        return input

    def build_desired(self, event):
        desired=[0]*self.max_event
```

```
        desired[event]=1
        return desired

    def normal(self,event):
        self.count=self.count+1
        w=self.check_window
        d,n=self.knowledge

        if not d.has_key(event):
            return 0

        input=[0]*self.max_event
        for i in range(len(w)):
            input[w[i]]=1.0/(2.0**i)
        desired=[0]*self.max_event
        desired[d[event]]=1

        out=array(n.calc(input))
        v=out[d[event]]
        out=clip(out-v,0,1)

        w.insert(0,d[event])
        if len(w) > self.window_size:
            del(w[-1])

        return max(out) < 0.1 and sum(out) < 1
```

7.6 Le super-analyseur

Il permet de démultiplexer les flux multiplexés en flux élémentaires et instancie un autre analyseur pour chaque flux élémentaire, comme on peut le voir dans le listing 7.8.

Listing 7.8 – Classe du super-analyseur

```
class SuperAnalyzer(Analyzer):
    name="SuperAnalyzer"
    parameters="-m Model [-M model_parameters] [-i]"
    def __init__(self,model=None,ignore_new_types=0,model_parm=[],options=[]):
        self.model=model
        self.ignore_new_types=ignore_new_types
        self.model_parm=model_parm
        Analyzer.__init__(self,options)
        self.knowledge={}
        self.flux={}

    def parse_options(self,options):
        try:
            opts=getopt.getopt(options,"m:M:")
        except getopt.error,errmsg:
            raise getopt.error,"%s model: %s" %(self.name,errmsg)

        if len(opts[1]) > 0:
```

```
            raise getopt.error,"%s model: Parameters unrocognized : %s etc."
                                % (self.name,opts[1][0])
        for opt,parm in opts[0]:
            if opt == "-m":
                try:
                    self.model=models[parm]
                except KeyError:
                    raise getopt.error,"unknown model : %s" % parm
            elif opt == "-M":
                self.model_parm=string.split(parm)
            elif opt == "-i":
                self.ignore_new_types=1


        if not self.model:
            raise getopt.error,"SuperAnalyzer: no model specified"


    def learn(self,event):
        self.count=self.count+1
        try:
            fluxid,type,evt=event
        except TypeError:
            fluxid,type,evt=1,1,event
        kn=self.knowledge
        if not kn.has_key(type):
            kn[type]=self.model(options=self.model_parm)
        if not self.flux.has_key(fluxid):
            self.flux[fluxid]=self.model(options=self.model_parm)
            self.flux[fluxid].set_knowledge(kn[type].get_knowledge())
        self.flux[fluxid].learn(evt)
        self.lastflux=fluxid
        self.lasttype=type

    def normal(self,event):
        self.count=self.count+1
        try:
            fluxid,type,evt=event
        except TypeError:
            fluxid,type,evt=1,1,event
        kn=self.knowledge
        if not kn.has_key(type):
            return self.ignore_new_types
        if not self.flux.has_key(fluxid):
            self.flux[fluxid]=self.model(options=self.model_parm)
            self.flux[fluxid].set_knowledge(kn[type].get_knowledge())
        return self.flux[fluxid].normal(evt)

    def stats(self):
        s=[]
        # Give stats only for the 5 first ones
        for key in self.flux.keys()[:5]:
            s=s+map(lambda x,key=key: "%12s %s" % (repr(key),x),
                    self.flux[key].stats())
```

```
        s.append("Total: %i events" % self.count)
        s.append("Total: %i flux of %i different types"
                 % (len(self.flux),len(self.knowledge)))

        return s

    def logs(self):
        return "%s %s %s"
               % (repr(self.lasttype),repr(self.lastflux),
                  self.flux[self.lastflux].logs())
```

Chapitre 8

Tests d'intrusion

8.1 Données de [FHSL96]

Le fichier `sendmail.daemon.int` est un fichier contenant les traces de processus `sendmail` en utilisation normale. Il est tiré des traces utilisées dans [FHSL96]. C'est un fichier texte dont la première colonne correspond au PID d'un processus et dont la seconde correspond à un numéro d'appel système. Pour un PID donné, la suite des appels systèmes correspond au déroulement du processus. On peut voir un extrait de ce fichier :

```
8840 2
8840 66
8840 66
8840 4
8840 138
```

Nous allons tout d'abord l'apprendre en utilisant le modèle de "Forrest" :

```
./eidf.py -l -d sendmail.log  -f sendmail.eidf            \
          -s ForrestDisk -S '-f smt/sendmail.daemon.int'  \
          -m SuperAnalyzer -M '-m Forrest'
```

Nous pouvons ensuite vérifier que la base de connaissance reconnaît bien ces mêmes traces ainsi que d'autres correspondant aussi à un comportement normal :

```
./eidf.py -c -f sendmail.eidf                             \
          -s ForrestDisk -S '-f smt/sendmail.daemon.int'  \
          -m SuperAnalyzer -M '-m Forrest'
Total: 1571584 events
Total: 147 flux of 1 different types
Alerts: 0

./eidf.py -c -f sendmail.eidf                  \
          -s ForrestDisk -S '-f smt/sm-10763'  \
```

```
          -m SuperAnalyzer -M '-m Forrest'
Total: 362 events
Total: 1 flux of 1 different types
Alerts: 0

./eidf.py -c -f sendmail.eidf                  \
          -s ForrestDisk -S '-f smt/sm-10814'  \
          -m SuperAnalyzer -M '-m Forrest'
Total: 374 events
Total: 1 flux of 1 different types
Alerts: 0
```

Si par contre on utilise des traces d'attaques ou de fonctionnement auquel l'analyseur n'a pas été habitué, on obtient les résultats suivants :

```
./eidf.py -c -f sendmail.eidf                        \
          -s ForrestDisk -S '-f smt/fwd-loop-1.int'  \
          -m SuperAnalyzer -M '-m Forrest'
Total: 635 events
Total: 2 flux of 1 different types
Alerts: 6

./eidf.py -c -f sendmail.eidf                            \
          -s ForrestDisk -S '-f smt/syslog-local-1.int'  \
          -m SuperAnalyzer -M '-m Forrest'
Total: 1575 events
Total: 6 flux of 1 different types
Alerts: 40

./eidf.py -c -f sendmail.eidf                            \
          -s ForrestDisk -S '-f smt/syslog-local-2.int'  \
          -m SuperAnalyzer -M '-m Forrest'
Total: 1517 events
Total: 6 flux of 1 different types
Alerts: 35
```

Conclusion

Discussion sur l'architecture

L'architecture ainsi développée répond au problème de modularité. Le développement de nouveaux modules, que ce soit des sources d'audit ou des analyseurs, est facile et rapide.

L'architecture ayant un but expérimental, on n'hésite pas, lors de l'implémentation, à sacrifier quelques optimisations pour coller au modèle ou à utiliser des solutions générales plutôt que des cas particuliers plus rapides. Certains paramètres comme la taille mémoire n'ont pas toujours été pris en compte alors que ce sont ces paramètres qui parfois ont décidé l'architecture de systèmes de détection d'intrusion. Il est alors parfois difficile d'utiliser une telle implémentation en dehors du cadre de la recherche, car ses performances peuvent soit ne pas suivre le comportement d'un système, soit le ralentir énormément. Cependant, une partie de la lenteur est inhérente au langage utilisé, et une telle architecture peut être reprise pour des produits en production.

Développements futurs

L'architecture proposée peut recevoir une infinité de nouvelles sources d'audit ou de nouveaux analyseurs. Parmi ceux qui n'ont pas été implémentés, on peut citer :

- une source d'audit pour écouter le trafic réseau[1]
- une source d'audit pour tracer plusieurs processus, capable de suivre les appels systèmes qui dupliquent le processus (`fork()`)
- une source d'audit renvoyant les exécutions effectuées par un processus
- une source d'audit utilisant les commandes passées aux shells
- une source d'audit utilisant le journal du système (syslog)

Une super-source pourrait aussi être créée, qui permettrait de placer plusieurs sources d'audit sur des systèmes différents. Ce serait un premier pas vers une architecture distribuée. Il manquerait bien sûr la redondance et la prise en compte des analyses d'une source par l'analyseur d'une autre source.

Aucun analyseur utilisant l'approche de détection de malveillances n'a été implémenté. Cela peut néanmoins parfaitement être fait. Même si une phase d'apprentissage est prévue, elle n'est pas obligatoire si par exemple l'analyseur apporte sa propre base de connaissances, dans ce cas une base de signatures d'attaques.

[1] ce module est partiellement implémenté

On peut aussi étendre cette architecture en lui rajoutant un module de réponse. Ce module pourrait par exemple prévenir la personne chargée de la sécurité par alarme, courier électronique, téléphone, pager, etc. On pourrait également rajouter des modules de contre-mesure, en commençant peut-être par s'inspirer de [SF00] ou de [JMKM99].

Enfin, un frontal graphique peut être également utilisé par dessus le programme.

Autres applications

Des applications autres que la détection d'intrusions peuvent tirer profit de cette architecture, comme la détection de fraude à la carte de crédit en utilisant les séquences d'achats précédentes, ou encore pour vérifier la conformité de certains produits sur des chaînes de montage. Plus généralement, toute application ayant pour but de détecter des anomalies ou des malveillances est concerné.

Bibliographie

[AFV95] D. Anderson, T. Frivold, and A. Valdes. Next-generation intrusion detection expert system, 1995. 30

[And80] J. Anderson. Computer security threat monitoring and surveillance, 1980. 7

[Axe98] S. Axelsson. Research in intrusion-detection systems : A survey, 1998. 28

[Bac99] Rebecca Bace. An introduction to intrusion detection and assessment. Prepared for ICSA.net., 1999. 8

[BFFI+98] Jai Sundar Balasubramaniyan, Jose Omar Farcia-Fernandez, David Isacoff, Eugene Spafford, and Diego Zamboni. An architecture for intrusion detection using autonomous agents. Technical report, November 1998. 24

[BK88] David S. Bauer and Michael E. Koblentz. NIDX – an expert system for real-time network intrusion detection. In *Proceedings of the Computer Networking Symposium*, pages 98–106, Washington, DC, 1988. IEEE Computer Society Press. 7

[Can98] J. Cannady. Artificial neural networks for misuse detection, 1998. 25

[CCD+99] S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, J. Rowe, S. Staniford-Chen, R. Yip, and D. Zerkle. The design of grids : A graph-based intrusion detection system, 1999. 33

[CDE+96] Mark Crosbie, Bryn Dole, Todd Ellis, Ivan Krsul, and Eugene Spafford. IDIOT - user guide. Technical report, September 1996. 32

[CG] Hung-Jen Chang and Joydeep Ghosh. Pattern association and retrieval in a continuous neural system.

[Clo] Jacques Clot. *La réponse imunitaire chez l'homme*. Documentation immunologique INAVA. 26

[DBS92] Hervé Debar, Monique Becker, and Didier Siboni. A Neural Network Component for an Intrusion Detection System. In *Proceedings of the 1992 IEEE Computer Society Symposium on Reserach in Security and Privacy*, pages 240–250. IEEE, IEEE Service Center, Piscataway, NJ, 1992. 34

[Den87] Dorothy E. Denning. An intrusion-detection model. *IEEE Transactions on software engeneering*, SE-13 :222–232, 1987. 7

[FH] Stephanie Forrest and Steven A. Hofmeyr. Immunology as information processing. 26

[FHS97] Stephane Forrest, Steven A. Hofmeyr, and Anil Somayaji. Computer immunology. *Communications of the ACM*, 40(10) :88–96, 1997. 26

[FHSL96] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for Unix processes. In *Proceedinges of the 1996 IEEE Symposium on Research in Security and Privacy*, pages 120–128. IEEE Computer Society Press, 1996. 2, 6, 48, 50, 55, 67

[Fin] Luca Deri Finsiel. Ntop : a lightweight open-source network ids.

[FJL+98] Teresa F.Lunt, R Jagannathan, Rosanna Lee, Sherry Listgarten, David L. Edwards, Peter G. Neumann, Harold S. Javitz, and Al Valdes. Ides : The enhanced prototype, a real-time intrusion detection system. Technical report, SRI Project 4185-010, SRI-CSL-88-12, CSL SRI Internationnal, Computer Science Laboratory, SRI Intl. 333 Ravenswood Ave., Menlo Park, CA 94925-3493, USA, October 1998. 29

[Gib84] William Gibson. *Neuromancer*. 1984. 16

[GS92] C. R Gent and C. P Sheppard. Predicting time series by a fully connected neural network trained by back propagation. *Computing and Control Engineering Journal*, May 92. 35

[Gus] Dug Song Gus. nidsbench - a network intrusion detection system test suite.

[HB95] L. Halme and R. Bauer. Aint misbehaving - a taxonomy of antiintrusion techniques, 1995. 11

[HF00] Steven A. Hofmeyr and Stephanie Forrest. Architecture for an artificial immune system. *Evolutionary Computation*, 8(4) :443–473, 2000. 26

[HFS98] S. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls, 1998.

[Hol75] John H Holland. *Adaptation un Natural and Artificial Systems*. PhD thesis, University of Michigan, Ann Arbor, 1975. 26

[HWHM98] G. Helmer, J. Wong, V. Honavar, and L. Miller. Intelligent agents for intrusion detection, 1998. 24

[HWHM00] G. Helmer, J. Wong, V. Honavar, and L. Miller. Lightweight agents for intrusion detection, 2000. 24

[IKP95] Koral Ilgun, Richard A. Kemmerer, and Phillip A. Porras. State transition analysis : A rule-based intrusion detection approach. *Software Engineering*, 21(3) :181–199, 1995. 31

[Ilg91] K. Ilgun. Implementation of ustat audit data preprocessor, 1991. 31

[Ilg93] Koral Ilgun. USTAT : A real-time intrusion detection system for UNIX. In *Proceedings of the 1993 IEEE Symposium on Research in Security and Privacy*, pages 16–28, Oakland, CA, 1993. 31

[JMKM99] W. Jansen, P. Mell, T. Karygiannis, and D. Marks. Applying mobile agents to intrusion detection and response, 1999. 24, 70

[JMKM00] W. Jansen, P. Mell, T. Karygiannis, and D. Marks. Mobile agents in intrusion detection and response, 2000. 24

[Lan] Terran Lane. Machine learning techniques for the domain of anomaly detection for computer security.

[LN97] D. Lane and P. Nolan. Application of pattern matching techniques to example based diagnosis. In R. A. Adey, G. Rzevski, and R. Teti, editors, *Applications of Artificial Intelligence in Engineering XII. [Full papers on CD ROM]*, pages 113–14. Comput. Mech. Publications, Southampton, UK, 1997.

[LNY+00] Wenke Lee, Rahul A. Nimbalkar, Kam K. Yee, Sunil B. Patil, Pragneshkumar H. Desai, Thuan T. Tran, and Salvatore J. Stolfo. A data mining and CIDF based approach for detecting novel and distributed intrusions. In *Recent Advances in Intrusion Detection*, pages 49–65, 2000. 23

[LS98] Wenke Lee and Salvatore Stolfo. Data mining approaches for intrusion detection. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, 1998. 23

[LSC97] Wenke Lee, Salvatore Stolfo, and Patrick Chan. Learning patterns from unix process execution traces for intrusion detection. In *Proceedings of the AAAI97 workshop on AI methods in Fraud and risk management*, 1997.

[LSM] Wenke Lee, Salvatore J. Stolfo, and Kui W. Mok. Algorithms for mining system audit data. 23

[LSM99] Wenke Lee, Salvatore J. Stolfo, and Kui W. Mok. A data mining framework for building intrusion detection models. In *IEEE Symposium on Security and Privacy*, pages 120–132, 1999. 23

[LTG+92] Teresa F. Lunt, Ann Tamaru, Fred Gilham, R. Jagannathan, Caveh Jalali, and Peter G Neuman. A real-time intrusion detection expert system (ides). Technical report, Project 6784, CSL, SRI Internationnal, Computer Science Laboratory, SRI Intl. 333 Ravenswood Ave., Menlo Park, CA 94925-3493, USA, February 1992. 29

[Lun88] Teresa F. Lunt. Automated audit trail analysis and intrusion detection : A survey. In *Proceedings of the 11th National Computer Security Conference*, Baltimore, MD, 1988.

[Lun90] Teresa F. Lunt. IDES : An Intelligent System for Detecting Intruders. In *Proceedings of the symposium : Computer Security, Threat and Countermeasures*, Rome, Italy, 1990. 29

[Mé] Ludovic Mé. Sécurité des systèmes d'information. Transparents de cours.

[Mé96] Ludovic Mé. Genetic algorithms , a biologically inspired approach for security audit trails analysis, 1996. 27

[Mé98] Ludovic Mé. Gassata, a genetic algorithm as an alternative tool for security audit trails analysis. In *Proceedings of the First International Workshop on the Recent Advances in Intrusion Detection, Louvain-la-Neuve, Belgium*, 1998. 27, 33

[MA96] Ludovic Mé and Véronique Alanou. Détection d'intrusion dans un système informatique : méthodes et outils. In *TSI*, volume 4, pages 429–450. 1996. 26

[NN00] Stephen Northcutt and Judy Novak. *Network Intrusion Detection. An Analyst's Handbook*. New Riders, 2000.

[NP89] Peter G. Neumann and Donn B. Parker. A Summary of Computer Misuse Techniques. In *Proceedings of the 12th National Computer Misuse Techniques*, 1989.

[PZC+96] Nicholas J. Puketza, Kui Zhang, Mandy Chung, Biswanath Mukherjee, and Ronald A. Olsson. A methodology for testing intrusion detection systems. *Software Engineering*, 22(10) :719–729, 1996.

[Roe99] Martin Roesch. Snort : Lightweight intrusion detection for networks. In *Proceedings of LISA '99*, pages 229–238, 1999.

[SCCC+96] S. Staniford-Chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, C. Wee, R. Yip, and D. Zerkle. Griids - a graph based intrusion detection system for large networks, 1996. 33

[SCCC+99] S. Staniford-Chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, C. Wee, R. Yip, and D. Zerkle. The design of grids : A graph-based intrusion detection system, 1999. 33

[SF00] Anil Somayaji and Stephanie Forrest. Automated response using system-call delays. In *Proceedings of the 9th USENIX Security Symposium*, 2000. 70

[Sím98] Jirí Síma. Introduction to neural networks. Technical report, institute of Computer Science, Academy of Sciences of the Czech Republic, 1998. 62, 76

[Sma88] Stephen E. Smaha. Haystack : An Intrusion Detection System. In *Fourth Aerospace Computer Security Applications Conference*, pages 37–44, Tracor Applied Science Inc., Austin, TX, 1988. 7, 28

[SSHW88] M. M. Sebring, E. Shellhouse, M. E. Hanna, and R. A. Whitehurst. Expert system in intrusion detection : A case study. In *Proceedings of the 11th National Computer Security Conference*, pages 74–81, 1988. 7, 29

[Syv94] Paul Syverson. A taxonomy of replay attacks. In *Computer Security Foundations Workshop VII*. IEEE Computer Society Press, 1994.

Annexes

Annexe A

Réseaux neuronaux

Pour plus de détails, on se réferrera à [Sím98].

A.1 Définition

Soit X l'ensemble des neurones d'entrée, Y l'ensemble des neurones de sortie. Soit w_{ij} le coefficient d'entrée du stimulus venant du neurone i dans le neurone j. Soit ξ_i l'excitation totale reçue par le neurone i. Soit y_i la sortie du neurone i. Soit σ_i la fonction de transfert du neurone i. Soient $j_{\leftarrow}$ l'ensemble des nœuds en entrée du nœud j et $j_{\rightarrow}$ l'ensemble des nœuds dont j est une entrée.

$$\xi_j = \sum_{i \in j_{\leftarrow}} y_i w_{ij} \tag{A.1}$$

$$y_i = \sigma_i(\xi_i) \tag{A.2}$$

A.2 Rétropropagation

A.2.1 Principe

On introduit l'erreur totale E sur une séquence d'apprentissage $(\overrightarrow{x}, \overrightarrow{d})$ où $\overrightarrow{d}$ est la sortie désirée lorsque l'entrée est $\overrightarrow{x}$:

$$E(\overrightarrow{w}) = \frac{1}{2} \sum_{j \in Y} [(y_j(\overrightarrow{w}, \overrightarrow{x}) - d_j]^2 \tag{A.3}$$

Si $\overrightarrow{w}^{(t)}$ est le vecteur des coefficients du réseau au temps discret t, on calcule ce vecteur au temps $t+1$ grace à :

$$\overrightarrow{w}^{(t+1)} = \overrightarrow{w}^{(t)} - \varepsilon \overrightarrow{\text{grad}} \left(E \left(\overrightarrow{w}^{(t)} \right) \right) \tag{A.4}$$

où $\varepsilon \in]0,1[$ est le taux d'apprentissage.

A.2.2 Calcul

En projetant l'équation A.4, on a :

$$w_{ij}^{(t+1)} = w_{ij}^{(t)} - \varepsilon \frac{\partial E}{\partial w_{ij}} \left(\overrightarrow{w}^{(t)} \right) \tag{A.5}$$

or

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial \xi_j} \frac{\partial \xi_j}{\partial w_{ij}} \tag{A.6}$$

En utilisant A.1, on a

$$\frac{\partial \xi_j}{\partial w_{ij}} = y_i \tag{A.7}$$

En utilisant A.2, on a

$$\frac{\partial y_j}{\partial \xi_j}(\xi_j) = \sigma_j'(\xi_j) \tag{A.8}$$

Soit $L_0 = Y$. On définit par récurrence L_k pour $k > 0$ par $L_k = \{ j \mid j_{\rightarrow} \subset L_{k-1} \}$. Il existe k_0 tel que L_{k_0} contient tous les nœuds car le graphe est acyclique.

Si $j \in L_0$ alors on a directement à partir de A.3

$$\frac{\partial E}{\partial y_j} = y_j - d_j \tag{A.9}$$

Si $j \in L_k - L_{k-1}$ avec $k > 0$, on suppose qu'on sait calculer $\frac{\partial E}{\partial y_i}$ pour tout $i \in L_{k-1}$. On a alors

$$\frac{\partial E}{\partial y_j} = \sum_{r \in j_{\rightarrow}} \frac{\partial E}{\partial y_r} \frac{\partial y_r}{\partial \xi_r} \frac{\partial \xi_r}{\partial y_j} = \sum_{r \in j_{\rightarrow}} \frac{\partial E}{\partial y_r} \sigma_r'(\xi_r) w_{rj} \tag{A.10}$$

On sait donc calculer $\frac{\partial E}{\partial y_i}$ pour tout $i \in L_k$.

A.3 Algorithme utilisé pour le perceptron multicouche

Soit c le nombre de couches du perceptron. On appelle $L_1 = \{N_{11}, \ldots, N_{1n_1}\}$ la couche des neurones d'entrée, $L_c = \{N_{c1}, \ldots, N_{cn_c}\}$ la couche des neurones de sortie, les $L_i, 1 < i < c$ étant les couches intermédiaires. On appelle $\overrightarrow{W_{ij}}$ le vecteur des coefficients d'entrée de N_{ij}.

Soit $\overrightarrow{x}$ le vecteur d'entrée et $\overrightarrow{y}$ le vecteur de sortie.

Soit $n_0 = \dim(\overrightarrow{x})$. On peut remarquer que

$$\forall i \in [1..c] \quad \forall (j,k) \in [1..n_i]^2 \quad \dim(\overrightarrow{W_{ij}}) = \dim(\overrightarrow{W_{ik}}) = n_{i-1}$$

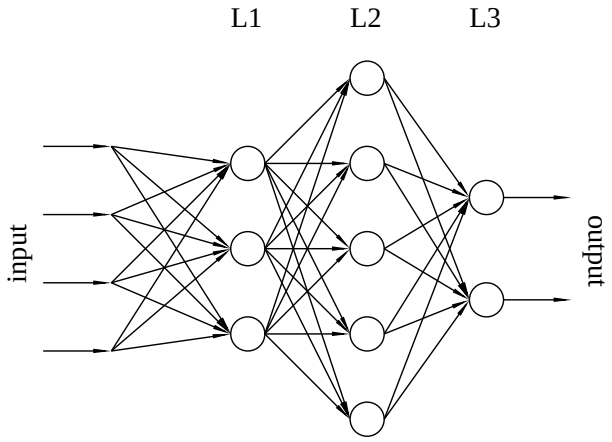

FIG. A.1 – Exemple de réseau de neurones

On appelle l_i la fonction qui, à un vecteur d'entrée $\overrightarrow{X_i}$ associe le vecteur de sortie $\overrightarrow{Y_i}$ tel que sa k^{ieme} coordonnée $Y_{ik} = \sigma_{N_{ik}}(\overrightarrow{X_i}.\overrightarrow{W_{ik}})$.

On a donc $\overrightarrow{y} = (l_c \circ \ldots \circ l_1)(\overrightarrow{x})$.

Alg. A.1 – Calcul de $\overrightarrow{y}$ par le réseau (algorithme de feedforward)

$\overrightarrow{X_0} \leftarrow \overrightarrow{x}$

for $k = 1$ to c **do**

  $\overrightarrow{X_k} \leftarrow l_k(\overrightarrow{X_{k-1}})$

end for

$\overrightarrow{y} \leftarrow \overrightarrow{X_c}$

Une fois $\overrightarrow{y}$ obtenu, on calcule l'erreur par rapport à la sortie désirée $\overrightarrow{d}$: $E = \frac{1}{2}\|y - d\|_2^2$. Connaissant son erreur e_{ij}, un neurone N_{ij} peut réajuster ses propres coefficients et peut communiquer aux neurones qui le précèdent la part de responsabilité que chacun avait pour son erreur, c'est-à-dire calculer un vecteur $\overrightarrow{r_{(i-1)j}}$ de dimension n_{i-1} tel que sa k^{ieme} coordonné $r_{(i-1)j_k}$ soit la responsabilité du neurone $N_{(i-1)k}$ dans l'erreur du neurone N_{ij}.

A.4 Implémentation

L'implémentation des réseaux neuronaux s'est également faite de façon orientée objet, comme on peut le voir sur le listing A.1.

Tout d'abord, un objet implémentant une fonction de transfert est créé. Il propose la méthode `sigma(x)` pour calculer cette fonction de transfert au point x et la méthode `dsigma(x)` pour calculer la valeur de sa dérivée au point x.

Lorsqu'un neurone est instancié, il prend en paramètre une instance de fonction de transfert et une liste de coefficients. Plusieurs neurones peuvent partager la même instance d'une fonction de transfert. La taille de la liste des coefficients fixe le nombre d'entrées du neurone. Il met ensuite à disposition une méthode pour calculer sa valeur d'activation en fonction de valeurs d'entrée et une méthode pour apprendre une valeur de sortie pour une liste de valeurs d'entrée donnée.

Pour instancier un réseau de neurones, il faut passer en paramètre une liste de listes d'instances de neurones, chaque liste de neurones étant une couche du perceptron. On doit bien évidement avoir comme condition que le nombre d'entrées de chaque neurone de la couche n est égal au nombre de neurones de la couche $n-1$. L'instance de réseau de neurone permet, comme un neurone, de calculer la sortie associée à une entrée donnée et également d'ajuster les coefficients de chaque neurone du réseau par rétro-propagation à partir d'une entrée et d'une sortie désirée.

Listing A.1 – Implémentation de réseaux neuronaux

```
from math import exp
import Gnuplot
from Numeric import *

################################################
# Classes to implement transfert functions
# to be loaded in a neuron

class TransfertFunc:
    def __init__(self):
        pass
    def sigma(self,x):
        return x
    def dsigma(self, x):
        return 1

class HardLimiter(TransfertFunc):
    def __init__(self):
        TransfertFunc.__init__(self)
    def sigma(self,x):
        return x >= 0
    def dsigma(self, x):
        # the dirac peak is missing !
        return x==0

class Sigmoid(TransfertFunc):
    def __init__(self,k=1):
        TransfertFunc.__init__(self)
        self.k=k
    def sigma(self,x):
        try:
            den=(1+exp(-self.k * x))
            return 1/den
        except OverflowError:
            print "Overflow in sigma. return 0"
            return 0
    def dsigma(self,x):
```

```
        try:
            den=((1 + exp(-self.k*x))**2)
            return self.k*exp(-self.k*x) / den
        except OverflowError:
            print "Overflow in dsigma. return 0"
            return 0


################################################
## class for a neuron

class Neuron:
    def __init__(self,transfertFunc,coefs,bias=0,learn_rate=0.1):
        self.coefs=array([bias]+coefs)
        self.tfunc=transfertFunc
        self.learn_rate=learn_rate

    def excitation(self, input):
        return cross_correlate(self.coefs,[1]+input)[0]

    def calc(self, input):
        return self.tfunc.sigma(self.excitation(input))

    def backpropagate(self, input, derr, error):
        xcit=self.excitation(input)
        dxcit=self.tfunc.dsigma(xcit)
        ret=(self.coefs*derr*dxcit)[1:].tolist()
        input=array([1]+input)
        self.coefs=self.coefs-self.learn_rate*derr*dxcit*input
        return ret


    def learn(self, input, desired):
        calc=self.calc(input)
        derr=self.calc(input)-desired
        return self.backpropagate(input,derr,0.5*derr**2)

################################################
## Class for a multilayer perceptron

class NeuralNet:
    def __init__(self,net):
        self.net=net

    def calc(self, input):
        res=input
        for layer in self.net:
            res=map(lambda neuron,inpt=res : neuron.calc(inpt),layer)
        return res


    def learn(self, input, desired):
        reslayer=[]
        res=input
```

```
        for layer in self.net:
            reslayer.append(res)
            res=map(lambda neuron,inpt=res : neuron.calc(inpt),layer)

        invnet=self.net[:]
        invnet=map(lambda x,y: (x,y), invnet,reslayer)
        invnet.reverse()

        res=map(lambda out,desir: out-desir,res,desired)

        E=0.5*reduce(lambda x,y:x+y,map(lambda x:x**2,res))

        for layer,resl in invnet:
            res=map(lambda neuron,err,inpt=resl,E=E: neuron.backpropagate(inpt,err,E),
                    layer,res)
            res=reduce(lambda x,y: map(lambda z,t:z+t,x,y),res)
        return res
```

Annexe B

Le modèle de Markov caché

Un modèle de Markov caché (HMM) est composé de
- un ensemble de N états $Q = \{q_1, \ldots, q_n\}$. L'état du système au temps t appelé s_t.
- une matrice de probabilité de transition d'états $A = (a_{ij})_{1 \leq i,j \leq N}$ ou a_{ij} est la probabilité que l'état s_{t+1} soit q_j si $s_t = q_i$, ie $a_{ij} = P(s_{t=1} = q_j | s_t = q_i)$.
- un ensemble de M symboles observables $V = \{v_1, \ldots, v_M\}$ et une distribution de probabilité discrète sur ces symboles pour chaque état. C'est-à-dire que la distribution b_j donne à un temps t la probabilité d'observer un symbole v_k quand le système est dans l'état q_j : $b_j(k) = P(v_k \text{ observé } | s_t = q_j)$
- une distribution initiale des état $\pi = (\pi_i)$ où $\pi_i = P(s_1 = q_i)$, exprimant la probabilité que le système commence à l'état q_i.

Un modèle de Markov caché λ est entièrement défini par les trois paramètres $\lambda = (A, B, \pi)$.

Il y a trois problèmes fondamentaux assiciés aux HMM :

Probabilité d'observation : étant donnés une séquence d'observations $O = O_1 \ldots O_T$ et un modèle λ, calculer la probabilité d'observer cette séquence d'observations avec ce modèle, $P(O|\lambda)$

Sélection de séquences d'états : étant donnés une séquence d'observations $O = O_1 \ldots O_T$ et un modèle λ, calculer la séquence d'états $s_1 s_2 \ldots s_T$ la plus probable (selon certains critères d'optimalité) de produire O.

Entrainement du modèle : étant donnés une séquence d'observations $O = O_1 \ldots O_T$, calculer le modèle λ qui maximise $P(O|\lambda)$

Annexe C

Le code : morceaux choisis

C.1 La charpente

Listing C.1 – Code de la charpente

```
#! /usr/bin/python

import sys,os
import getopt,string


import analyzers
import audit

LEARN=1
CHECK=2

DEFAULT_DBFILE="profile.eidf"

################################################
## Parameters analysis

def usage():
    print "Usage: %s {-l|-c} [-a] [-h] [-d|D dumpfile] [-t threshold] [-v] [-u]\n" \
          "       [-s audit-source] [-S 'source-parameters']\n" \
          "       [-m model] [-M 'model-parameters'] [-f knowledge-file]" % sys.argv[0]
    print "Available Models:"
    for m in analyzers.models.keys():
        print "  * %s: %s" % (m,analyzers.models[m].parameters)
    print "Available audit sources:"
    for s in audit.sources.keys():
        print "  * %s: %s" % (s,audit.sources[s].parameters)


try:
    opts=getopt.getopt(sys.argv[1:],"ahvulact:d:D:s:S:m:M:f:")
```

```
mode=0
verbose=0
append=0
dbfile=DEFAULT_DBFILE
model=analyzers.models.keys()[0]
mparm=""
source=audit.sources.keys()[0]
sparm=""
dumpfile=""
dumpmode="w"
multiflux=1
threshold=1

for opt,parm in opts[0]:
    if opt == "-h":
        usage()
        sys.exit(0)

    if opt == "-v":
        verbose=verbose+1

    if opt == "-u":
        multiflux=0

    if opt == "-t":
        try:
            threshold=float(parm)/100
            if threshold < 0 or threshold > 100:
                raise getopt.error,"bad threshold : %i. Must be between 0 and 100" % threshold
        except ValueError,str:
            raise getopt.error,"threshold : %s" % str


    if opt == "-d":
        dumpfile=parm
        dumpmode="w"

    if opt == "-D":
        dumpfile=parm
        dumpmode="a"

    if opt == "-a":
        append=1

    elif opt == '-l':
        if mode == CHECK:
            raise getopt.error,"-l and -c are exclusive"
        mode=LEARN

    elif opt == '-c':
        if mode == LEARN:
            raise getopt.error,"-l and -c are exclusive"
        mode=CHECK
```

```
        elif opt == "-f":
            dbfile=parm

        elif opt == "-m":
            if not analyzers.models.has_key(parm):
                raise getopt.error,"model %s deos not exist" % parm
            model=parm

        elif opt == "-s":
            if not audit.sources.has_key(parm):
                raise getopt.error,"source %s deos not exist" % parm
            source=parm

        elif opt == "-S":
            sparm=sparm+" "+parm

        elif opt == "-M":
            mparm=mparm+" "+parm

    if len(opts[1]) > 0 :
        raise getopt.error,"Parameters unrocognized : %s etc." % opts[1][0]

    if mode == 0:
        raise getopt.error,"-l|-c parameter is mandatory"

    #Instantiate the model class
    analyzer=analyzers.models[model](options=string.split(mparm))

    #Instantiate the source class
    auditsrc=audit.sources[source](options=string.split(sparm))



except getopt.error,errmsg:
    print "*** Error:", errmsg
    print
    usage()
    sys.exit(1)

#####################################################
## Begin the analysis

if dumpfile:
    try:
        dmp=open(dumpfile,dumpmode)
    except IOError,(errno,msg):
        print "Error: [errno=%i] while opening %s: %s" %(errno,dumpfile,msg)
        sys.exit(4)

if mode ==CHECK or append:
    try:
        analyzer.load_knowledge(dbfile)
    except ValueError,errmsg:
        print "Error:",errmsg
```

```
        sys.exit(2)
    print "Knowledge loaded from file %s" % dbfile


try:
    auditsrc.open()
except ValueError,errmsg:
    print errmsg
    sys.exit(3)

alerts=0
try:
    if verbose:
        stats=analyzer.stats()
        print "\n"*len(stats),
    while analyzer.completed() < threshold:
        event=auditsrc.getevent()
        if not multiflux:
            if type(event)==type(()):
                if len(event)==3:
                    event=event[2]
        if mode == LEARN:
            analyzer.learn(event)
        elif mode == CHECK:
            if not analyzer.normal(event):
                alerts=alerts+1
        if dumpfile:
            dmp.write(analyzer.logs()+"\n")
        if verbose==1:
            stats=analyzer.stats()
            stats.append("Alerts: %i" % alerts)
            print "\033[A"*(len(stats)+1)
            print string.join(stats,"      \n")
        elif verbose==2:
            print event


except KeyboardInterrupt:
    print "Stopped by user."
except EOFError,msg:
    print msg
except ValueError,msg:
    print msg


if dumpfile:
    dmp.close()


if mode == LEARN:
    analyzer.save_knowledge(dbfile)
    print "Knowledge saved in file %s" % dbfile
auditsrc.close()
```