N° d'ordre : 411 Année 2005

THÈSE

présentée en vue de
l'obtention du titre de

DOCTEUR
de
L'ÉCOLE NATIONALE SUPÉRIEURE
DE L'AÉRONAUTIQUE ET DE L'ESPACE

ÉCOLE DOCTORALE : Informatique et télécommunications
SPÉCIALITÉ : Programmation et systèmes
par

Fabien AUTREL

Fusion, corrélation pondérée et réaction dans un environnement de détection d'intrusions coopérative

Soutenue le 7 mars 2005 devant le jury :

MM.	R.	DEMOLOMBE	Président
	S.	BENFERHAT	
Mme	A.	CAVALLI	
MM.	F.	CUPPENS	Directeur de thèse
	H.	DEBAR	Rapporteur
	B.	LE CHARLIER	Rapporteur

Thèse préparée au sein du Département traitement de l'information et modélisation de l'ONERA-centre de Toulouse.

Remerciements

Je remercie tout d'abord l'ONERA et plus particulièrement René Jacquart ainsi que Jacques Cazin, son successeur à la direction du DTIM, pour m'avoir accueilli au sein de leur laboratoire pendant les deux premières années de ma thèse. Je remercie Jacques pour m'avoir permis de finir ma thèse avec Frédéric Cuppens lors de sa prise de poste à l'ENST bretagne. Je remercie l'antenne de Rennes de l'ENST Bretagne pour m'avoir acueilli pour la dernière année de ma thèse et en particulier Xavier Lagrange et Gilbert Martineau.

Je remercie Robert Demolombe pour avoir accepté de présider mon jury ainsi que mes rapporteurs, Hervé Debar et Baudoin Le Charlier. Je remercie aussi Ana Cavalli pour avoir accepté de faire partie du jury.

Je remercie énormément Frédéric Cuppens pour avoir été mon directeur pendant trois ans. Merci à toi Frédéric pour m'avoir supporté, pour avoir survécu à toutes les épreuves que j'ai pu te faire subir. Je te remercie de m'avoir fait découvrir le monde de la recherche, après la courte introduction qu'aura été mon DEA, et de m'avoir donné envie de continuer dans cette voie. Merci pour ta gentillesse et pour la confiance que tu me témoignes. J'éspère continuer à travailler avec toi, de près ou de loin.

Je remercie Salem Benferhat pour avoir été mon co-directeur de thèse. J'ai énormément apprécié nos discussions, en rapport ou non avec ma thèse, qui m'ont fait découvrir une personne très cultivée. J'éspère aussi continuer à travailler avec toi.

Merci à Nora Cuppens pour avoir lu et re-lu et re-re-lu mon rapport de thèse et pour toutes les suggestions, les virgules et les corrections ! J'apprécie tes conseils et ton expérience.

Merci à Christiane Payrau et à Noëlle Desblancs pour leur gentillesse et leurs compétences qui m'ont beaucoup aidé, merci à Monique Peron pour son aide précieuse lors de la gestion de la fin de ma thèse à distance. J'adresse un grand merci à toutes les personnes que j'ai rencontré à l'ONERA de Toulouse, Pierre Biebert, Claire Saurel, Patrice Cros, Josette Brial et toutes les autres personnes qui ont contribué à la richesse de ces trois années de thèse.

Merci à Rémi et Nawel, mes compagnons de galère émmigrés de Poitiers. Vous rencontrer à l'ENSMA a été le début d'une grande amitié. Merci à toi Rémi de m'avoir donné envie de jouer de la guitare, ne t'en fais pas, je sais que mon médiocre niveau n'est pas en relation avec les cours que tu m'as donné. Merci à François Carcenac, le plus jazzy des thésards de l'ONERA, je te souhaite tout le bonheur du monde ainsi qu'à ta petite famille, Amandine et Matthieu. Merci à Christophe "capuche" Garion, te voir autrement qu'en tenue de hardcore est toujours un plaisir. Merci à toi pour ton amitié, je n'oublierai jamais ce que mes oreilles ont entendu grâce à toi. Un grand salut à tous les thésards que j'ai croisé, Jérôme, Jérôme "Erminator", Laurent, Bruno, Laurent "qualité", Benjamin, Rania, Pilar et tous les autres que j'oublie de citer. Merci à Renaud, le plus Toulousain des développeurs d'émulateurs.

Tu es la seule personne avec qui je peux parler d'émulation sans avoir l'impression d'être un geek ;). Merci à Thierry et Alexandre, les deux thésards les plus sympas de la Bretagne.

Je remercie ma famille pour m'avoir suivi pendant ces trois années, vous m'avez toujours soutenu dans mes choix qui m'ont fait prendre le chemin de la recherche. Merci d'être venu de Bretagne me voir dans mes petits souliers le jour de ma soutenance ! J'ai une pensée toute particulière pour mon grand-père paternel. Merci à vous tous de me donner tant, et ce depuis que je suis né.

Un grand salut à tous mes amis, aucune relation d'ordre n'existant dans cet ensemble : Alain, Marion, Benoît, Dorothée, Guillaume, Charles, Pascal, Damien, Katell, Damien, Françoise, Vincent, Céline, Sébastien, Erwann, François, Hubert, François-Marie, Rosen, Matthieu, Benjamin, Stefano et tous ceux qui se reconnaissent comme faisant partie de cet ensemble.

Enfin je remercie celle qui m'a soutenu depuis la deuxième année de ma thèse, celle qui illumine ma vie depuis lors, Aline, mon amour. J'éspère que tu me supporteras encore longtemps, le plus longtemps possible...

Table des matières

Chapitre 1

Introduction

L'environnement des systèmes informatiques, alimenté par les nouvelles technologies et une interconnexion croissante par le biais d'Internet, est en constante évolution. La sécurité informatique consiste à assurer un fonctionnement correct d'un système dans cet environnement, qui se révèle souvent malveillant. Plus précisément, le but poursuivi est de faire en sorte qu'un système informatique respecte les propriétés suivantes [42] :

- confidentialité : propriété d'une information qui n'est ni disponible, ni divulguée aux personnes, entités ou processus non autorisés [ISO 7498-2]. Exprimer les besoins de confidentialité, c'est déterminer les utilisateurs autorisés et la limite de leurs prérogatives. Formuler des exigences en matière de confidentialité d'une information revient à énoncer des critères sur lesquels se fonde la légitimité des accès à cette information :
  - critères liés à la personne : identité, appartenance à un groupe, habilitation etc .
  - critères liés à la fonction : droits, autorisations, besoin d'en connaître, besoin d'en user, etc.
  - critères liés à un rôle : responsabilités, délégations, nécessités, etc.

- disponibilité : La disponibilité ou fiabilité de service est la prévention d'un déni non autorisé d'accès à l'information ou à des ressources (critères communs [ISO/IEC 15408]). Elle doit pouvoir garantir que les tâches critiques en temps sont exécutées au moment voulu. Il s'agit également de garantir que l'accès aux ressources est possible quand on en a besoin, et que les ressources ne sont pas sollicitées ou conservées inutilement.

- intégrité : propriété assurant que des données n'ont pas été modifiées ou détruites de façon non autorisée [ISO 7498-2]. L'intégrité consiste à garantir un maintient correct entre les relations spécifiques des différentes données et l'échange des données entre utilisateurs ou processus sans avoir subi d'altération. L'intégrité rassemble les fonctions suivantes :
  - Les fonctions destinées à garantir que des données n'ont pas été modifiées d'une manière non autorisées,

- Les fonctions permettant de déceler ou d'empêcher toute perte, ajout ou modification lorsque les données sont échangées,
- Les fonctions interdisant la modification de la source ou de la destination du transfert de données.

Le respect de ces propriétés par un système informatique se fait par deux approches : L'approche prévention et l'approche détection. Nous présentons successivement ces deux approches puis nous verrons comment elle sont liées.

1.1 La prévention

La prévention de problèmes et/ou d'attaques permettant de violer les propriétés de sécurité d'un système informatique se fait à travers la définition d'une politique de sécurité. Cette politique de sécurité utilise les résultats d'une analyse de risque effectuée sur le système à sécuriser. L'expression de la politique de sécurité se fait grâce à différents modèles se basant sur le contrôle d'accès ou le contrôle de flux de données. Parmis les modèles de contrôle d'accès, on peut citer :

- les modèles discrétionnaires (Discretionary Access Control : DAC, [60]) . DAC est un modèle de contrôle d'accès où les utilisateurs du système peuvent autoriser ou interdire l'accès aux objets qu'ils contrôlent à d'autres utilisateurs.
- les modèles à base de rôles (Role Based Access Control : RBAC, [41, 80]) . Le modèle RBAC base les décisions de contrôle d'accès sur les fonctions qu'un utilisateur possède au sein d'une organisation.
- les modèles à base d'organisations et de rôles (Organisation Based Access Control : ORBAC, [61]). Ce modèle est centré sur l'organisation, permet l'expression de privilèges contextuels et introduit la notion d'obligation, qui en fait un modèle de contrôle d'usage outre le contrôle d'accès.

Les modèles de ontrôle de flux sont essentiellement des modèles de type obligatoires (Mandatory Access Control : MAC, [79]). Le modèle MAC permet de restreindre l'accès aux objets en se basant sur le niveau de confidentialité des informations contenues dans les objets et les autorisations des utilisateurs à accéder à une information d'un certain niveau de confidentialité.

L'application de cette politique, si elle est correctement conçue, permet le respect des propriétés de sécurité du système. Les mécanismes permettant l'application d'une politique de sécurité se répartissent sur plusieurs composants du système. Par exemple un système fonctionnant sous Linux permet de définir des utilisateurs, des groupes d'utilisateurs, des droits pour chaque groupe ou utilisateur, etc... Le système d'exploitation d'une machine fournit donc généralement un ensemble de mécanismes permettant l'application de la politique de sécurité ou d'une partie de celle-ci. Un autre moyen d'appliquer une politique est le pare-feu. La politique de sécurité définit alors des règles de filtrage utilisées pour configurer le pare-feu.

La définition d'une politique de sécurité n'est pas chose facile et la difficulté croit avec la complexité de l'organisation à la base de cette politique. Des défauts de conception peuvent être introduits et ainsi produire une politique ne garantis-

sant plus les propriétés de sécurité du système. Ces failles de sécurité peuvent être exploitées par une personne malveillante.

1.2 Limites de la prévention

Un système informatique ne peut pas être sécurisé seulement par la définition d'une politique de sécurité. En effet un système informatique comprend toujours des failles de sécurité dont les mécanismes de contrôle d'accès ne peuvent empêcher l'exploitation.

Les vulnérabilités peuvent permettre à une personne mal intentionnée d'attaquer un système afin d'en violer la politique de sécurité. Ces vulnérabilités sont introduites à plusieurs niveaux lors de la conception et l'implantation d'un système informatique. Mais elles peuvent également apparaître dans la conception et la mise en œuvre de la politique associée au système. Dans ce cas le traitement des résultats de la détection d'intrusions peut alimenter l'approche prévention en permettant d'améliorer la politique de sécurité et son déploiement.

Les vulnérabilités introduites au moment de l'implantation d'une application ne peuvent être évitées. En effet, étant donnée la complexité croissante des applications, la réalisation d'un logiciel dépourvu de tout bug est chose impossible. Ces bugs dans l'implantation de logiciels peuvent être exploitées de diverses manières. Parmi ces vulnérabilités, la plus courante est la possibilité de réaliser un dépassement de capacité d'un buffer (buffer overflow) permettant d'exécuter un programme arbitraire, le plus souvent avec les privilèges les plus élevés. Le programme exécuté pourra rendre la machine indisponible (Déni de Service, *DOS* en anglais), installer un cheval de Troie permettant de prendre le contrôle de la machine à distance ou installer un programme esclave permettant de relayer du spam ou utiliser la machine pour une attaque distribuée.

Un autre type de vulnérabilité réside dans la faiblesse de certains protocoles. Par exemple le protocole TCP offre la possibilité d'envoyer des paquets de demande de fin de connexion, ce qui permet par exemple de fermer illégalement une connexion. Afin d'empêcher l'envoi arbitraire de paquets de demande de fin de connexion pour interrompre une connexion légale, un numéro de session TCP est associé à chaque paquet. Cependant Paul A. Watson [87] a montré que l'utilisation de la technique de "fenêtre glissante" permet l'acceptation d'une série de numéros dans une session TCP. Cette vulnérabilité augmente considérablement les chances de réussite d'une attaque.

1.3 La détection d'intrusions

Étant donné l'impossibilité d'éviter la présence de vulnérabilités dans un système informatique, il est nécessaire de définir des mécanismes détectant l'utilisation de ces vulnérabilités. La détection de l'utilisation de vulnérabilités de sécurité peut aboutir à une correction de ces vulnérabilités, si cela est possible, ou peut mener à

une réaction visant à arrêter la progression de l'attaque.

La détection d'intrusions se propose de détecter l'exploitation de failles de sécurité. Nous allons tout d'abord voir les techniques utilisées pour détecter ces exploitations puis nous verrons qu'il est nécessaire de raisonner sur le flux d'événements généré par la détection.

1.3.1 Les techniques de détection

La détection d'exploitation de vulnérabilités dans les systèmes se fait par des composants logiciels ou matériels. Par détection de l'exploitation de vulnérabilités nous désignons le processus de détection d'événements correspondant à l'exploitation de ces vulnérabilités. Cependant la détection d'intrusions ne se limite pas à la détection d'événements liés à l'exploitation des vulnérabilités ; elle détecte aussi des événements suspects. Ces événements sont suspects dans la mesure où ils sont souvent associés à d'autres événements correspondants à l'exploitation de failles de sécurité. De manière générale, ces composants sont appelés sondes de détection d'intrusions (SDI). Un SDI surveille soit une machine ou alors surveille le trafic réseau [85]. Les SDI ne modifient pas le flux d'informations qu'ils observent contrairement à un firewall par exemple.

Les techniques de détection peuvent être classées en deux grandes familles :

- Approche par signature : le SDI reçoit un flux d'informations et compare ces données avec une base de signatures.
- Approche comportementale : le SDI mesure plusieurs variables et détecte un comportement suspect lorsque ces variables s'écartent des valeurs d'un profil représentant un comportement normal.

Ces deux approches ont leurs avantages et leurs inconvénients respectifs.

1.3.2 Approche comportementale

L'approche comportementale a été la première technique à être utilisée dans le domaine de la détection d'intrusions [8]. Cette approche se base sur des modèles statistiques afin d'identifier un comportement suspect dans le système surveillé. Un SDI comportemental peut surveiller une application ou bien un utilisateur.

Les sujets observés par un SDI comportemental sont caractérisés par leur profil statistique qui représente leur comportement. Les profils statistiques des utilisateurs et des applications sont calculés en mesurant certaines variables du système. Ces variables peuvent être par exemple l'utilisation du réseau, les accès au disque dur, les fichiers accèdes, etc. Le profil statistique du sujet à surveiller (utilisateur du système ou application) est calculé sur une période pendant laquelle le sujet observé a une activité considérée comme normale [31, 67]. Ce calcul est une phase, dite phase d'apprentissage, très importante car le profil résultant représente le comportement normal du sujet. Ensuite, dans la phase de surveillance par le SDI, un profil statistique représentant le comportement réel du sujet est calculé en temps réel et comparé au profil calculé dans la phase d'apprentissage. Cette comparaison se fait

généralement à travers le calcul d'une valeur réelle représentant l'écart entre le profil mesuré et le profil de référence [57]. De plus certains SDI actualisent lentement le profil de référence pour prendre en compte l'évolution du comportement du sujet sur une longue période.

Cette approche présente plusieurs avantages. Le premier et le principal est que ce type de SDI est en théorie capable de détecter de nouvelles attaques [40, 57]. C'est principalement pour cette raison que cette technique a été introduite. Le deuxième avantage de cette approche est le fait de ne pas avoir à expliciter le comportement anormal d'un sujet. Il suffit d'observer le sujet pendant une phase de comportement normal afin de pouvoir mettre en œuvre la détection.

1.3.3 Approche par signature

L'approche par détection de signatures a été la seconde technique de détection d'intrusions à être implantée après l'approche comportementale. Les premiers algorithmes se basaient sur le principe de comparaison de chaînes de caractères, le SDI cherchant alors une séquence d'octets dans un paquet réseau permettant d'identifier un trafic malveillant. Les systèmes plus récents analysant le trafic réseau inspectent aussi les champs de données des paquets en fonction du protocole utilisé.

Le SDI a accès à une base de signatures décrivant les motifs que l'on veut détecter dans le flux d'information reçu par le SDI. Ces motifs peuvent être par exemple une certaine séquence d'octets pour un SDI surveillant le trafic réseau (NIDS) ou une certaine séquence d'appels système pour un SDI surveillant une machine (HIDS).

Un SDI utilisant l'approche par signature a potentiellement un taux de faux positifs faible pourvu que sa base de signature soit correctement spécifiée. La qualité de ce type de SDI se juge surtout par la base de signature qui lui est associée.

Il existe plusieurs SDI utilisant l'approche par signature, certains étant des produits commerciaux (NFR[5], RealSecure[4], Cisco Secure[2], Dragon[1]) et d'autres gratuits(Snort[82], Prelude[3], SHADOW[6]). Les SDI auxquels les utilisateurs peuvent ajouter des signatures ont l'avantage de pouvoir bénéficier des contributions des personnes de la communauté de la sécurité informatique. Snort est par exemple un IDS gratuit dont la base de signature est régulièrement mise à jour.

1.3.4 Limites et problèmes

Les différentes approches de détection d'intrusions ont chacune leurs limites et leur utilisation pose des problèmes.

L'approche par signature ne permet pas de détecter des attaques non spécifiées dans la base de signature. La probabilité pour qu'une nouvelle attaque puisse être détectée par la signature d'une autre attaque est très faible. De ce fait ce type de SDI ne peut pas détecter de nouvelles attaques et nécessite une mise à jour régulière de la base de signatures. Pour les attaques connues, il est nécessaire de spécifier correctement la signature. En effet il arrive qu'un sous-ensemble de la séquence de motifs constituant la signature soit présent dans le flux d'information reçu par le SDI,

et que ce sous-ensemble ne corresponde pas à une occurrence de l'attaque spécifiée par la signature. On voit donc clairement ici qu'il est très important de bien spécifier la signature d'une attaque, une sous-spécification entraînant la génération d'alertes ne correspondant pas à des occurrences de l'attaque. Ce type d'alerte générée lors de la détection d'événements ne correspondant pas à l'occurrence d'attaques associées aux signatures est appelé *faux positif*. En contrepartie, une signature trop contrainte ne permet pas de détecter toutes les occurrences de l'attaque associée à une signature. Cette absence d'alerte est appelé *faux négatif*.

Un autre problème de ce type de SDI réside dans la granularité des alertes. Les événements détectés correspondent généralement à des actions ou des attaques de granularité très faible. Par exemple Snort est capable de détecter l'occurrence d'un balayage du port 139 d'une machine, qui en soit ne constitue pas une action malveillante, mais qui peut être associée à d'autres actions constituant une tentative de violation de la politique de sécurité. Ainsi il est possible d'observer par exemple l'action *winnuke*[1] après un balayage de port car ce dernier aura permis à un attaquant de savoir que la machine balayée héberge une version de Windows vulnérable à cette attaque.

Dans le cas de SDI se basant sur une méthode statistique, le principal problème réside dans le fait qu'il est très difficile d'expliquer pourquoi la mesure statistique du comportement du sujet observé dévie de la normale. Il devient alors difficile de diagnostiquer une anomalie détectée par un SDI comportemental. Le second problème de ce type de SDI est la quantité de faux positifs générés, la déviation du comportement du sujet observé pouvant être dû à une légère modification de la nature de ses activités. D'autre part, un attaquant se sachant observé par ce type de SDI pourra modifier très lentement son comportement afin qu'il soit appris (nous avons dit dans la section 1.3.2 que certains algorithmes mettent à jour le profil statistique de référence pour prendre en compte une évolution dans le comportement d'un sujet). Dans ce cas une alerte ne sera pas émise alors que l'attaquant réalise son attaque (faux négatif).

On constate que l'utilisation d'un seul SDI ne permet pas de correctement surveiller un système. Un SDI réseau observant le trafic peut ne pas observer l'intégralité du trafic en fonction de son emplacement dans le réseau. Une base de signatures incomplète ne permettra pas de détecter toutes les attaques. De plus la granularité faible des alertes émises par les SDI ne permet pas d'avoir une vision globale de ce qui se passe dans le système.

Afin de pallier ses problèmes inhérents aux techniques de détection utilisées, il est nécessaire de faire coopérer les sondes. C'est cette approche multi-sonde que nous avons adoptée. L'utilisation de SDI se basant sur différentes bases de signatures permet d'augmenter la capacité de détection du système. La distribution des sondes à travers les composants du réseau permet de surveiller une plus grande partie du système. Il est ainsi possible, en utilisant les informations contenues dans plusieurs alertes, d'obtenir une vision plus globale d'une intrusion que si l'on analyse

[1] L'action *winnuke* permet de réaliser un déni de service sur une machine utilisant le système d'exploitation de Microsoft

séparément chaque alerte. Cette approche faisant coopérer plusieurs SDI est appelée *détection d'intrusions coopérative*.

Nous ne considérons que le cas de la coopération de SDI à base de signatures. En effet, les différentes techniques de traitement des alertes présentées dans cette thèse requièrent que les événements détectés soient nommés. Ainsi, il est possible d'identifier les événements détectés et de leur associer des propriétés. Plus précisément, nous utilisons des modèles de ces événements décrits dans un langage por exprimer leurs propriétés.

1.4 Plan de la thèse

Cette thèse est organisée de la manière suivante : le second chapitre présente en détail la détection d'intrusions coopérative, propose un état de l'art des architectures coopératives existantes, présente l'architecture retenue au cours de cette thèse ansi que les langages de modélisation d'attaque et les formats d'alerte existants. Les chapitres 3, 4 et 5 présentent respectivement les notions d'agrégation et de fusion, de corrélation et finalement de réaction aux intrusions. Chacun des chapitres 3, 4 et 5 commence par un état de l'art des différents travaux de recherche existants et détaille ensuite l'approche que nous avons développée. Le sixième chapitre présente les expérimentations que nous avons réalisées grâce aux implantations des notions d'agrégation, de fusion, de corrélation et de réaction définies dans les précédents chapitres. Enfin le septième chapitre conclut cette thèse et présente les perspectives de recherches ouvertes par ce travail.

Chapitre 2

Détection d'intrusions coopérative

Dans ce chapitre nous allons exposer le principe de la détection d'intrusions coopérative, nous verrons quels sont les travaux existants, l'architecture qui a été mise en place dans le cadre de cette thèse, puis nous étudierons les langages de modélisation des attaques ainsi que les formats d'alertes.

2.1 Principes

Nous avons vu dans le chapitre précédent qu'étant données les limites des SDI existants, il est nécessaire de faire coopérer un ensemble de SDI afin d'améliorer la qualité du diagnostic. Plus précisément, il s'agit de raisonner sur les alertes générées par un ensemble de sondes distribuées dans le système informatique surveillé. Afin de créer ce flux d'alertes, les alertes générées sont collectées et centralisées. Cette approche coopérative fait apparaître de nouvelles problématiques liées au traitement des alertes de détection d'intrusions générées par les sondes :

- Redondance des alertes : Lorsqu'une action non autorisée par la politique de sécurité est exécutée dans le système surveillé, un sous-ensemble des SDI la détecte en fonction du placement de chacun dans le réseau. Plusieurs alertes peuvent donc être générées par l'occurrence d'une action.

- Corrélation des alertes : une attaque contre le système peut se résumer à une seule action, par exemple l'exploitation d'une faille de sécurité permettant l'exécution d'un buffer overflow pour réaliser un déni de service. Dans ce cas, du point de vue de la détection d'intrusions, chaque SDI capable de détecter l'attaque générera normalement une seule alerte. De manière générale une attaque contre un système se base sur l'exploitation de failles de sécurité. L'exploitation de ces failles peut se faire au hasard, par exemple en attaquant un ensemble d'adresses IP en espérant que cet ensemble d'adresses contient l'adresse d'une ou de plusieurs machines vulnérables. Cependant l'attaquant peut aussi procéder de manière plus intelligente et essayer de récupérer des informations sur le système visé. Dans ce cas l'attaque se traduit par un scénario d'actions élémentaires du point de vue de l'attaquant, les premières actions visant à col-

lecter des informations sur le système ciblé et les actions suivantes représentant l'exploitation de ces informations. On voit bien ici que du point de vue de la détection d'intrusions, les alertes correspondant aux actions exécutées par l'attaquant s'organisent en un scénario d'alertes. Ces alertes sont donc reliées entre elles par des liens de corrélation. Ici le mot corrélation désigne un lien d'influence positif entre deux actions. Une action A et une action B sont corrélées quand l'exécution de l'action A favorise l'exécution de l'action B. Par favoriser nous désignons le fait que l'exécution de A peut modifier l'état du système de manière à ce qu'une condition nécessaire pour l'exécution de B soit vérifiée.

A partir de ces deux problématiques nous pouvons déjà identifier deux fonctionnalités importantes pour un système de détection d'intrusions coopératif, à savoir l'agrégation et la fusion d'alertes redondantes et la corrélation d'alertes.

2.1.1 Agrégation et fusion d'alertes

Afin de traiter la redondance des alertes dans un système de détection d'intrusions coopératif, une des fonctionnalités du système doit être de pouvoir déterminer si deux alertes ont été générées lors de la détection du même événement. Si l'on considère le flux d'alertes généré par l'ensemble des SDI, on peut imaginer un premier module de traitement ayant pour but de former des groupes d'alertes, les différentes alertes du groupe étant toutes relatives à l'occurrence d'un même événement. Ces groupes d'alertes sont souvent appellés clusters d'alertes.

Une fois les clusters d'alertes formés, les informations représentées par l'ensemble des alertes constituant un cluster doivent être fusionnées afin d'obtenir une alerte dite de fusion. L'étape de fusion permet de réduire la quantité de données nécessaire pour stocker le cluster d'alertes sans perdre d'informations.

Ce module de traitement des alertes n'ajoute pas d'information au flux d'alertes mais a pour but de réduire le nombre d'alertes à traiter sans en éliminer. Le fait de devoir comparer des alertes provenant de SDI hétérogènes et de fusionner leurs informations pose le problème du format des alertes. Cette question est abordée plus loin dans la section 2.5.

2.1.2 Corrélation d'alertes

La notion de corrélation d'alertes a été définie de manière différente par plusieurs travaux (voir par exemple [28]). Nous ne définissons pas formellement la notion de corrélation d'alertes dans cette section mais nous nous attachons à spécifier les fonctionnalités d'un module de corrélation d'alertes dans le contexte de la détection d'intrusions coopérative. Un tel module tente d'extraire du flux d'alertes des informations relatives aux liens existant entre certaines alertes du flux. Le but est d'extraire des scénarios d'alertes représentant les stratégies d'attaques employées par les personnes attaquant le système.

On se rend compte rapidement en énonçant la fonctionnalité désirée que chercher des liens de corrélation dans un flux d'alertes comportant des alertes redondantes

pose un problème si ces alertes dupliquées ne sont pas regroupées. Il apparaît donc nécessaire de coupler les fonctionnalités d'agrégation/fusion et de corrélation.

Le fait de pouvoir extraire un scénario d'attaque d'un ensemble d'alertes est intéressant dans le cadre d'une analyse hors ligne d'un ensemble d'alertes représentant par exemple une journée de surveillance d'un système. Constater que l'exécution d'un scénario est possible sur un système donné permet éventuellement de corriger des erreurs de conception ou de mise en œuvre du réseau. Dans ce cas, on reste dans le cadre d'une détection passive. Au lieu de constater l'occurrence d'un scénario, il est bien plus intéressant de pouvoir empêcher ce scénario. Ce type de traitement doit donc avoir lieu en temps réel au fur et à mesure que les alertes sont reçues. On identifie ainsi d'autres fonctionnalités intéressantes :

- reconnaissance des intentions de l'attaquant : lorsque le début d'un scénario est observé, il est intéressant de pouvoir prévoir les actions qui peuvent succéder à ces observations. En effet, pouvoir présenter à l'administrateur système l'ensemble des scénarios d'intrusion en cours d'exécution et surtout les suites possibles de chaque scénario permet d'augmenter la qualité du diagnostic.

- réaction aux intrusions : lorsqu'une action A est observée et que l'on est capable de déterminer que l'action B peut-être réalisée après A, si A fait partie d'un scénario dont le but est un objectif d'intrusion, il est souhaitable de pouvoir empêcher la réalisation de l'action B afin de contrecarrer l'avancement du scénario.

Avant d'aborder de manière plus formelle les notions d'agrégation, de fusion et de corrélation, intéressons-nous aux systèmes et architectures existants utilisant un ensemble de SDI répartis dans le système surveillé.

2.2 État de l'art

Dans cette section nous exposons les travaux existants présentant une architecture de détection faisant coopérer plusieurs SDI. Ce type d'architecture inclut généralement un module destiné à concentrer les informations générées par les SDI sur une seule machine. Cependant il existe aussi un autre type d'architecture appelée détection d'intrusions distribuée. Dans ce type d'architecture, le traitement des informations générées par les SDI est ditribué sur plusieurs ordinateurs afin de répartir la charge de calcul. Malgré cette différence, un des objectifs de ce type d'architecture est d'exploiter la puissance de détection d'un ensemble de SDI par rapport à celle d'une seule sonde.

Nous ne présentons que les choix d'architectures et les raisons motivant ces choix, nous ne rentrons pas dans le détail des algorithmes utilisés pour traiter les informations émanant de plusieurs SDI. Ces algorithmes peuvent se répartir en deux grandes catégories : les algorithmes destinés à fusionner les informations collectées et les algorithmes essayant de corréler ces informations afin de trouver des liens de causalité entre les alertes.

Les architectures de détection d'intrusions faisant intervenir plusieurs SDI peuvent se répartir en deux approches : l'approche distribuée qui vise à répartir les calculs d'analyse des informations générées par les SDI sur l'ensemble des SDI et l'approche coopérative qui centralise les données générées par les SDI pour les analyser.

2.2.1 Approche distribuée

Dans cette section nous passons en revue plusieurs travaux effectués dans le cadre de l'élaboration d'architecture de détection d'intrusions distribuée. Ce type d'architecture vise à distribuer la charge de calcul nécessaire pour l'analyse des informations récoltées par les sondes.

Dans [52], Huang et Wicks présentent une architecture utilisant des agents. Leur approche se focalise sur l'analyse des intentions de l'attaquant. D'après eux le contexte de génération d'une alerte est une donnée très importante. Ce contexte est notamment constitué par l'état de la machine surveillée par un SDI ou par l'ensemble des personnes connectées à un serveur. Ce contexte est dépendant du temps et peut varier très rapidement. La quantité de données à traiter en temps réel pour un système centralisé étant très importante, ils argumentent que la récupération de certaines informations du contexte de génération d'une alerte peut être rendue impossible par la latence du système. En effet étant donné que le contexte évolue rapidement, le temps de prendre la décision de récupérer certaines informations peut laisser le temps à ces informations de disparaître ou d'être modifiées. En conséquence, ils proposent d'utiliser des agents distribués afin de diminuer cette latence. Ils suggèrent une modélisation des intentions de l'attaquant utilisant les arbres-but. Un arbre-but est un arbre dont la racine est le but principal et dont les nœuds inférieurs sont des sous-buts à atteindre pour atteindre le but principal. Une version étendue de ces arbres est utilisée pour introduire la notion d'ordre (*ET* ordonné dans la figure 2.1). La figure 2.2 représente la détection d'un scénario en cours de réalisation. La réalisation des événements non détectés doit être vérifiée pour confirmer la réalisation du scénario. D'après les auteurs les intentions d'un attaquant sont susceptibles de changer plusieurs fois au cours d'une tentative de pénétration d'un système. En effet, si l'attaquant n'a pas de connaissance sur le système visé, il commencera par exécuter des actions lui permettant d'obtenir des informations sur la configuration du système et les éventuelles failles de sécurité présentes.

L'architecture du système de détection distribué est constituée par un ensemble d'agents locaux et un ensemble d'agents centraux. Un protocole de dialogue entre les agents locaux et centraux est défini. Il permet à un agent local d'annoncer à l'ensemble des agents centraux qu'il est disponible. Un agent central peut recruter un agent local disponible. Un agent local rend compte des scénarios, partiels ou complets, localement observés aux agents centraux qui l'ont recruté. Ces agents centraux sont chargés de fusionner les informations collectées afin de fournir à l'administrateur système un compte rendu de chaque tentative d'intrusion détectée. Les agents locaux peuvent aussi dialoguer entre eux. Par exemple un agent local peut demander

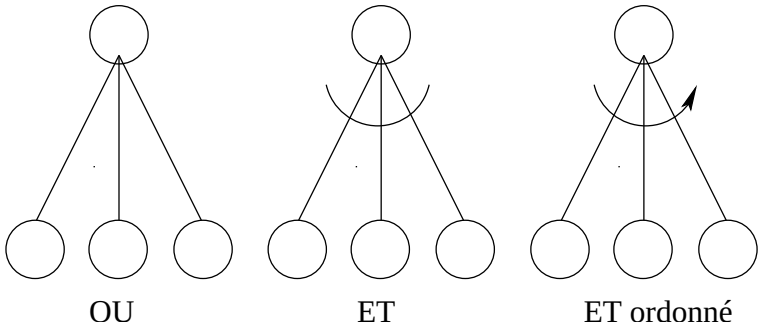

FIG. 2.1 – arbre-but *OU*, *ET* et *ET ordonné*

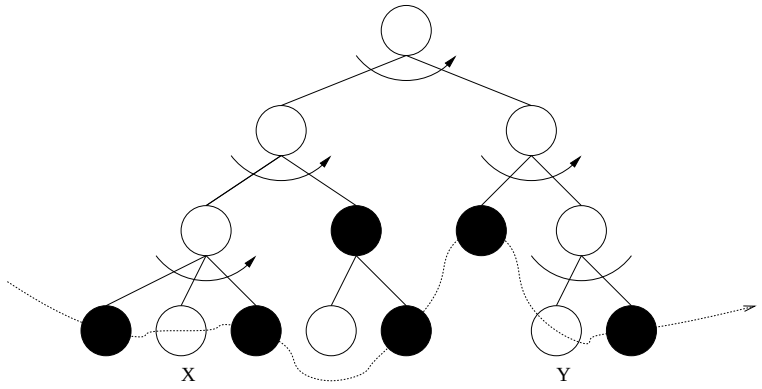

FIG. 2.2 – Scénario incomplet en progression. Les cercles noirs représentent les actions observées et les cercles vides les actions ou sous-buts non satisfaits.

des informations sur un événement à un autre agent local. Une des contraintes de ce type d'approche est la spécification d'une bibliothèque de scénarios suffisamment exhaustive pour assurer un taux de détection le plus élevé possible. La maintenance d'une telle bibliothèque n'est pas chose aisée car sa mise à jour peut être fastidieuse. Les auteurs ne précisent pas comment les informations sont fusionnées à travers la hiérarchie d'agents et ne proposent pas de modèle des informations circulant dans la hiérarchie.

Dans [48], Gopalakrishna et Spafford définissent une architecture distribuée sans hiérarchie d'analyse des données. Dans l'approche précédente un agent central est chargé de réunir les informations des agents distribués mais ici les auteurs proposent des agents qui s'occupent de faire toutes les analyses sur les données récupérées localement. Cette approche a pour avantage de décentraliser les calculs effectués sur les alertes et permet de répartir la charge sur plusieurs machines. Cependant les auteurs ne présentent dans cet article qu'une architecture, aucune information n'est donnée sur les modèles de donnée ou sur les algorithmes utilisés.

L'approche distribuée est séduisante en théorie car son principal but est de distribuer la charge de calcul liée à l'analyse des données sur plusieurs machines. Cependant les travaux que nous avons cités ne proposent que peu ou pas de formalisation des protocoles de dialogue et ne donnent pas de modèle de données pour les informations circulant sur le réseau. D'autre part, toutes les approches présentées ne sont pas implantées.

2.2.2 Approche coopérative

Cette section expose les travaux existants dans le domaine de la détection d'intrusions coopérative.

Dans [47], les auteurs soulèvent le problème de la fusion de flux de données provenant de plusieurs SDIs distribués dans le système surveillé. Ils présentent le système SCYLLARUS qui est une architecture de détection d'intrusions coopérative. L'architecture de ce système est présentée dans la figure 2.3.

Les auteurs utilisent la base de données orientée objet *CLASSIC* ([13, 14]) afin de modéliser les différentes entités du système surveillé. Par exemple, le concept SOFTWARE-VERSION représentant les informations sur la version d'un logiciel, est décrit de la manière suivante :

```
(AND CLASSIC-THING
(ALL major-version INTEGER)
(ALL minor-version INTEGER)
(ALL patchlevel INTEGER)
(ALL build INTEGER)
(AT-MOST 1 minor-version)
(AT-MOST 1 patchlevel)
(AT-LEAST 1 major-version)
(AT-MOST 1 major-version))
```

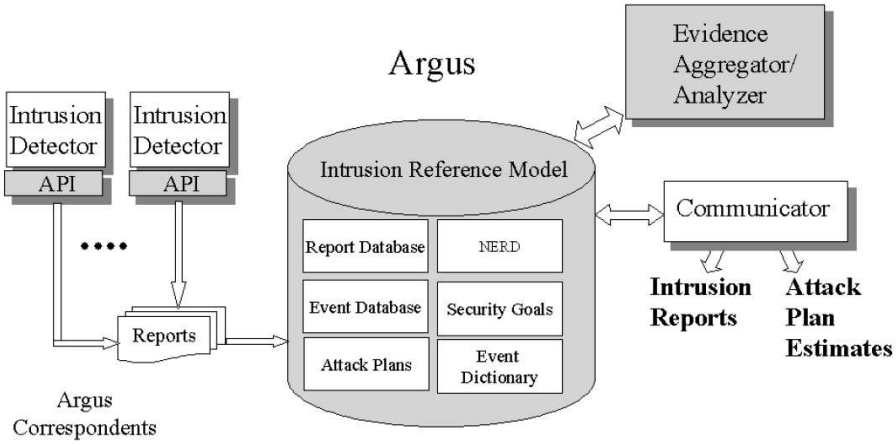

FIG. 2.3 – Architecture du système SCYLLARUS

Les relations d'agrégation entre les attributs (*major-version*, *minor-version*, *patchlevel* et *build*) et la classe SOFTWARE-VERSION sont spécifiées par les expressions *AT-MOST* et *AT-LEAST*. L'architecture du système se divise en composants dynamiques (la base de données de comptes-rendus, la base de données d'événements et la base de données de statut des objectifs d'intrusion) et composants statiques (la base de données sur les entités du réseau (NERD) et la base d'objectifs d'intrusion, le dictionnaire d'événements et la librairie de plans d'attaque). Les auteurs précisent que leur architecture est implantée et a été testée avec des événements générés artificiellement, c'est-à-dire ne correspondant pas à des alertes réelles. Il est intéressant de noter l'effort des auteurs pour introduire dans leur architecture une base de connaissances sur les propriétés du système surveillé. Cependant, il n'existe pas de détails sur les algorithmes utilisés pour faire coopérer les sondes.

Dans [43], Deborah Frincke et col. présentent une architecture permettant de mettre en place une plate-forme de détection d'intrusions coopérative. Ils définissent les relations pouvant exister entre les différentes entités du système et se basent dessus pour définir la politique de coopération :

- Peer : relation entre deux machines de deux domaines différents. Les machines ne se contrôlent pas mutuellement mais peuvent échanger des données. Deux machines peer ne se font pas forcément confiance.

- Manager : un manager donne des instructions à un ensemble de machine afin de leur préciser quelles données collecter, quand générer une alerte, etc. Un manager définit une politique de sécurité pour un ensemble de machine.

- Subordinate/Managed Host : c'est une machine qui reçoit tout ou partie de ses données et de sa politique de transmission de l'extérieur. Une telle machine

peut modifier ou ajouter des données à une politique et peut gérer d'autres machines. Une machine contrôlée doit faire confiance à la machine qui la contrôle.

- Slave Host : c'est une machine qui reçoit tout ou partie de ses données et de sa politique de transmission depuis l'extérieur. Une telle machine ne peut modifier ou ajouter des données à une politique mais peut gérer d'autres machines.

- Friend : relation entre deux machines du même domaine. Les machines ne se contrôlent pas mutuellement mais peuvent échanger des données. Deux machines friend se font confiance.

- Symbiote : relation caractérisant deux machines interdépendantes. Les machines ne se contrôlent pas mutuellement mais peuvent échanger des données. Deux machines partageant cette relation sont 'identiques' en terme de politique de sécurité appliquée.

Les auteurs proposent plusieurs principes que le système coopératif doit satisfaire :

- Contrôle de la politique locale de sécurité : une machine doit prendre ses décisions en fonction de la politique locale de sécurité. Même si cette machine fait partie d'une hiérarchie ou d'une relation maître/esclave, la légitimité de chaque interaction doit être vérifiée à partir de la politique de sécurité locale.

- Récupération autonome et collective des données : les données à analyser et à partager sont déterminées localement. Mais il se peut que des données soit collectées sur ordre d'un manager ou d'un peer et que ses données ne soient pas pertinentes vis à vis de la politique de sécurité locale à faire respecter.

- Confiance dans les données : les prises de décision basées sur des données venant de l'extérieur doivent prendre en compte un degré de confiance sur ces données.

- Respect de la politique et détection des violations : le mécanisme assurant le respect d'une politique et le mécanisme permettant la détection des violations d'une politique doivent être séparés. Une machine peut détecter une violation d'une politique de sécurité d'un domaine autre que le sien.

- Transactions sécurisées : il est nécessaire que les machines coopérantes s'authentifient mutuellement.

- Structure du partage des données : le partage de l'information peut se faire de manière verticale (par exemple entre un manager et une machine qu'il contrôle) ou horizontale (par exemple entre deux machines peer).

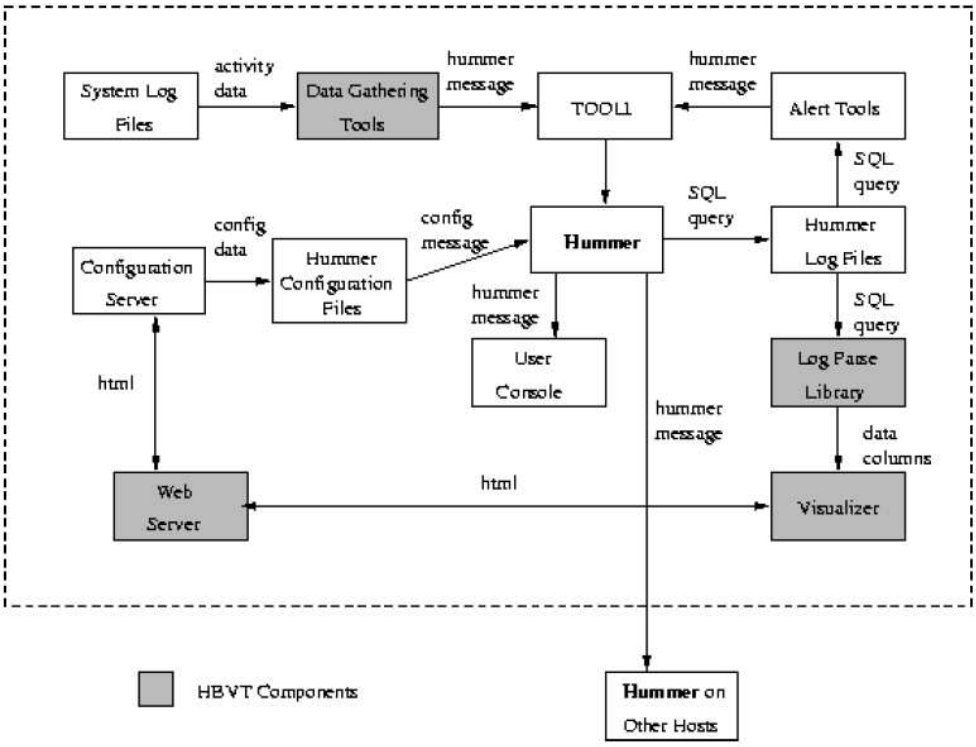

FIG. 2.4 – Interaction entre les composants d'Hummingbird.

– Stockage des données : les machines qui collectent les données doivent être capables de réduire la quantité de données collectées en éliminant les données redondantes. Elles doivent aussi "assainir" ces données.

– Outils de gestion du système et de visualisation : une interface utilisateur doit permettre de configurer facilement le système et doit fournir une visualisation simple des données collectées et des rapports.

Les auteurs proposent une implantations partielle des relations et des principes exposés appelée Hummingbird. La figure 2.4 représente les composants d'une instance d'Hummingbird ainsi que leurs interactions. La fonction de réduction/assainissement des données est implémentée par filtrage. Le protocole utilisé pour faire communiquer les agents est le protocole HMNR. Ce protocole permet le partage des données relatives à la sécurité entre les agents. Plusieurs prototypes ont été réalisés à petite échelle ([73, 54, 9, 39]) et combinés pour réaliser le prototype Hummingbird. Le protocole Kerberos ([74]) est utilisé pour l'authentification des agents Hummer entre eux, l'authentification entre les agents et leur base de données et le chiffrage des données. On peut regretter l'absence de spécification du format des données échangées ainsi que d'exemples d'application du système pour détecter des attaques connues. Bien que les auteurs présentent des arguments pour justifier leur architec-

ture à travers la définition de relations, présentées plus haut, et de propriétés, aucun exemple concret ne permet de se rendre compte de la pertinence de l'architecture. Cependant ce système représente avec le système EMERALD, que nous présentons juste après, un des premiers travaux sur une architecture complète de détection d'intrusions basée sur l'utilisation de plusieurs SDI.

Dans [76], Phillip A. Porras et Peter G. Neumann proposent une architecture de détection d'intrusions coopérative appelée EMERALD (Event Monitoring Enabling Responses to Anomalous Live Disturbances). EMERALD utilise une approche hiérarchique à trois couches et se destine à une utilisation dans les systèmes de grande taille. Chacune des trois couches est constituée d'agents. Ces agents sont appelés moniteurs dans l'architecture d'EMERALD. Nous verrons un peu plus loin en quoi consistent ces moniteurs. Chaque moniteur a ses propres détecteurs qui peuvent utiliser des méthodes statistiques et des méthodes à base de règles. Les 3 couches sont : la couche service (au plus bas niveau), la couche domaine et la couche entreprise (au plus haut niveau). La couche service n'observe qu'un seul domaine. Les agents de la couche domaine acceptent des données provenant de plusieurs agents de la couche service afin de pouvoir détecter des intrusions distribuées sur plusieurs domaines de service. De la même manière, les agents de la couche entreprise acceptent les données provenant de plusieurs agents de la couche domaine et tentent de détecter des attaques à l'échelle du système tout entier. L'analyse distribuée effectuée au travers du déploiement des agents fournit une abstraction globale de la coopération de plusieurs domaines du réseau. Les agents sont capables de s'échanger des informations produites par l'analyse des infrastructures (des routeurs par exemple) et des services (i.e les systèmes interfacés au réseau). Afin de limiter le trafic sur le réseau généré par ces échanges de données, un système d'abonnement permet à un moniteur de demander qu'un autre moniteur lui communique le résultat de ses analyses.

L'architecture générique d'un moniteur EMERALD (voir figure 2.5) permet en théorie l'interfaçage d'analyseurs basés sur d'autres méthodes que les méthodes statistiques et basé sur un système de règles.

Les données utilisées pour les analyses proviennent de flux d'événements dérivés de sources variées. Ces sources peuvent être des données d'audit, des datagrammes réseau, du traffic SNMP, des logs d'application et des résultats d'analyses faites par d'autres IDS. Les flux d'événements sont par nature hétérogènes et doivent être formatés avant d'être soumis à une analyse.

Le composant statistique du moniteur (*profilerengine*) fait une analyse basée sur la même approche que NIDES à partir d'un flux d'événement ([55]).

Le composant effectuant une analyse à base de signatures utilise une variante du système expert P-Best ([64]).

Les résultats des analyses de ces deux systèmes ainsi que ceux des autres analyseurs interfacés au moniteur sont soumis au solveur (*resolver*) pour être corrélés. C'est ce composant du moniteur qui est chargé d'appliquer la politique de réponse aux attaques. Les contre-mesures sont définies dans le champ *response-methods* d'un *ResourcesObject* (RO, figure 2.6).

Un des points les plus importants de la conception d'EMERALD est l'abstraction

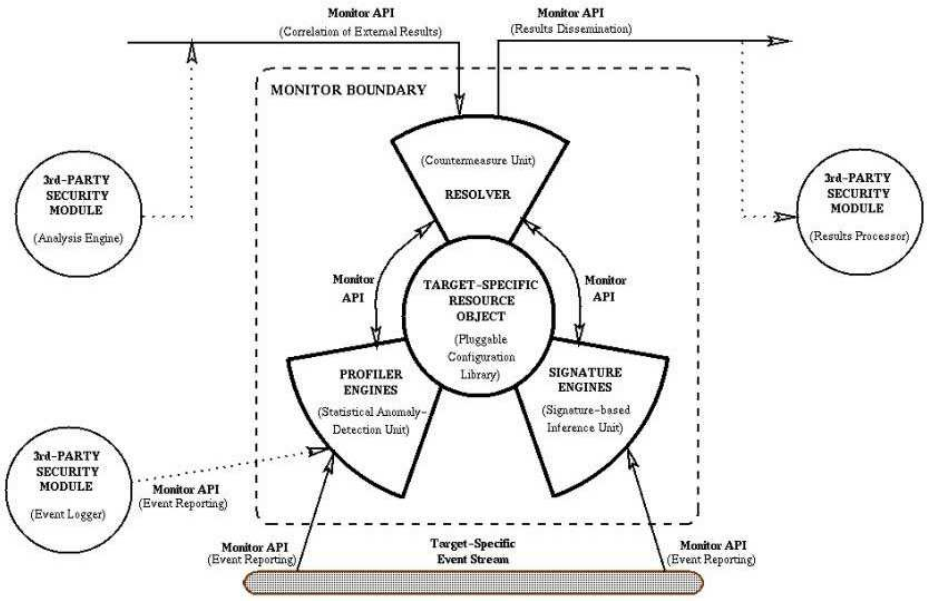

FIG. 2.5 – Architecture générique d'un moniteur EMERALD.

de la technique d'analyse d'un flux de données. Les *ResourcesObject* d'EMERALD contiennent tous les paramètres de chaque composant d'un moniteur ainsi que les techniques d'analyse (par exemple les mesures de l'anaylseur statistique ou les signatures du composant basé sur un système de règles) nécessaires pour traiter chaque type de flux associé à un élément du réseau. L'intérêt de ces objets est de définir une interface de dialogue et une architecture commune pour tous les moniteurs distribués dans le système à surveiller.

Un RO est composé des éléments suivants :

- Configurable event structure : définit une syntaxe universelle pour spécifier la structure d'un flux de données (c'est-à-dire l'ensemble des événements associés).

- Event-collection method : ensemble de routines de filtrage permettant à un analyseur de formater le flux de données provenant d'un type d'élément du réseau (cela peut être des logs applicatifs ou des paquets collectés sur le réseau par exemple).

- Engine N configuration : ensemble de variables et de structures de données qui spécifient la configuration d'un analyseur du moniteur.

- Analysis unit configuration : cette structure contient les variables définissant les sémantiques employées par un analyseur pour traiter un flux spécifique de

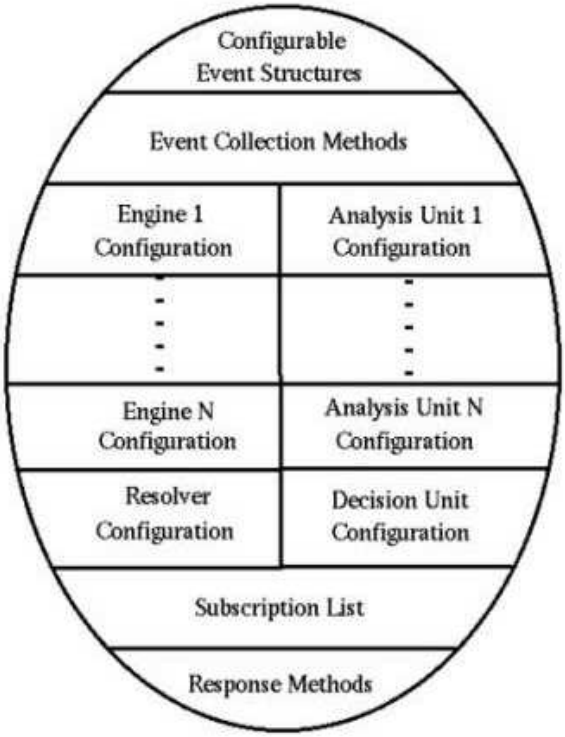

FIG. 2.6 – Architecture d'un resources object

données.

– Resolver configuration : spécifie la configuration des modules internes du solveur (*resolver*).

– Decision unit configuration : spécifie les techniques utilisées par le solveur pour fusionner les résultats de chaque analyseur.

– Subscription list : contient les informations d'authentification nécessaires à l'établissement des communications inter-moniteur.

– Response methods : ensemble des contre-mesures disponibles pour le solveur.

Les éléments d'un RO sont modifiés par des composants internes et par des clients externes autorisés à travers l'API du moniteur. Dans la hiérarchie d'analyse du système EMERALD, seuls les parents immédiats d'un moniteur peuvent modifier sa configuration.

Le système EMERALD propose une architecture intéressante de par son analyse hiérarchique des données permettant de déployer un ensemble de SDI dans un réseau. Cependant les auteurs ne présentent que très succinctement la façon dont les agents dialoguent et donnent peu de détails sur les techniques de détection utilisées. Dans [65] les auteurs présentent l'outil utilisé dans EMERALD pour détecter des intrusions à partir de signatures, *P-Best*. Le composant permettant de faire de la détection d'intrusions comportementale est le système $NIDES$ [55].

2.3 Architecture proposée

Nous proposons une architecture de type coopératif où les alertes produites par les sondes de détection sont centralisées dans une base de données. La centralisation des alertes se fait par une application, appelée concentrateur d'alertes sur la figure 2.7, qui insère les alertes dans la base de données. Ce même concentrateur est chargé de distribuer les alertes aux modules de traitement venant se connecter à lui. Chaque module se connectant au concentrateur peut spécifier les types d'événements qu'il veut recevoir. Ainsi chaque module peut ne traiter qu'une partie du flux d'alertes généré par les sondes.

Nous avons identifié deux modules de traitement des alertes correspondant aux fonctionnalités présentées dans la section 2.1. Le premier module est appelé module d'agrégation et de fusion d'alertes. Le second module, qui prend en entrée le résultat du traitement des alertes par le module d'agrégation et de fusion, est appelé module de corrélation d'alertes. La figure 2.7 représente l'architecture retenue.

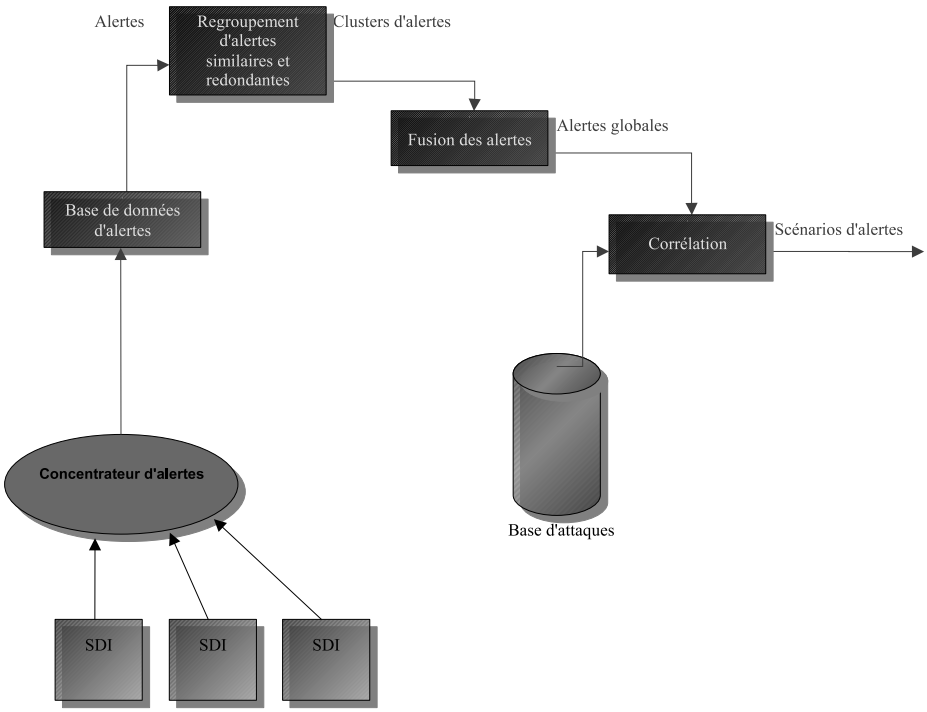

FIG. 2.7 – Architecture proposée pour la détection d'intrusions coopérative

2.3.1 Module d'agrégation et de fusion d'alertes

Ce module est chargé de regrouper les alertes similaires pour former des groupes d'alertes similaires, ou clusters d'alertes. Avant d'envoyer les groupes d'alertes au module de traitement suivant, ce module génère pour chaque groupe d'alertes similaires une alerte de fusion, dite alerte globale. Sur la figure 2.7 nous avons représenté un module d'agrégation et un module de fusion mais dans l'implantation ces deux fonctionnalités sont regroupées dans un même module.

2.3.2 Module de corrélation

Ce module est chargé de trouver les liens de corrélation existant entre les événements détectés par les SDI. Sa fonction est de reconnaître les ensembles d'alertes organisées en scénarios d'attaque, d'anticiper les intentions de l'attaquant afin de réagir avant qu'une attaque ne soit complètement réalisée. Ces informations sont présentées à l'administrateur dans une interface graphique afin de proposer une représentation intuitive des événements en cours de réalisation sur le système. Le moteur de corrélation s'occupe aussi de déterminer les contre-mesures efficaces pour chaque scénario détecté et propose à l'administrateur un ensemble d'actions permettant d'empêcher la progression ou d'annuler l'effet d'un scénario d'attaque sur le système. Le module de corrélation traite un flux d'alertes globales. En effet, étant donnée la complexité des calculs liés au traitement de la corrélation entre alertes, il

est important de traiter un nombre minimal d'alertes.

Nous avons vu précédemment un ensemble d'architectures dont le but principal est d'augmenter la qualité de la détection d'intrusions à travers l'utilisation de plusieurs SDI. Cependant nous n'avons pas abordé ni les algorithmes ni les modélisations associées à ces architectures. Nous allons donc maintenant nous intéresser aux langages définis pour décrire les événements détectés sur le système.

2.4 Langages de description d'attaques

Les SDI sont chargés de détecter l'occurrence de certains événements dans le système surveillé. Ces événements correspondent à des actions exécutées par l'attaquant, ces actions faisant partie de sa stratégie d'attaque. Une alerte correspondant à la détection d'une certaine action contient des informations sur les machines impliquées, c'est-à-dire la source de l'action et la (les) machine(s) ciblée(s), et donne le nom de l'action associée. La quantité et le type d'information transportée par l'alerte sont directement dépendants de la technique de détection utilisée. Ces informations ne nous renseignent pas sur la sémantique de l'action ; tenter de raisonner seulement sur ces informations est donc très difficile. Afin de pouvoir raisonner sur un ensemble d'alertes et tirer des conclusions de ces observations, il est nécessaire de modéliser les actions détectables afin d'associer une sémantique aux alertes. Dans ce qui suit, nous allons nous interesser aux langages de description d'attaques qui existent et comment ils sont utilisés.

Par *langage de description d'attaques*, nous désignons l'ensemble des langages permettant de modéliser la stratégie d'un attaquant. Cette modélisation peut se faire à différents niveaux de granularité. En effet, un langage peut être destiné à modéliser des scénarios d'actions constituant des scénarios d'intrusion (par exemple STATL [38]), ou peut-être plutôt destiné à la modélisation des actions élémentaires disponibles pour l'attaquant (par exemple LAMBDA [24]).

Nous verrons tout d'abord les langages orientés modélisation de scénarios puis nous verrons les langages orientés modélisation d'actions.

Cette classification en deux grandes familles de langages n'est pas stricte dans le sens ou certains langages permettent la modélisation d'actions élémentaires et la modélisation de scénarios complexes à partir de ces modèles élémentaires. Une classification possible des langages est donnée dans [69]. Les auteurs proposent la classification suivante :

- Langages de description d'*exploits* : ces langages permettent de décrire une attaque de manière à pouvoir la rejouer sur une machine.
- Langages de description d'événements : ces langages décrivent le format des événements utilisés dans le processus de détection.
- Langages de détection : ces langages permettent de spécifier les événements se manifestant lors de l'exécution d'une attaque sur un système.
- Langages de corrélation : ces langages permettent la modélisation des actions disponibles pour l'attaquant et permettent de raisonner sur les alertes afin de les grouper lorsqu'un lien de corrélation est trouvé.

- Langages de compte rendu : ces langages décrivent le format des alertes générées par les SDI.
- Langages de réponse : ces langages permettent la description de contre-mesures permettant de réagir lors de la détection d'une attaque.

Nous ne verrons pas ici les langages d'expression de signatures d'attaque spécifiques aux SDI tels que *Snort*, *BRO*, $P-Best$, etc.

Langages de modélisation de scénarios

- STATL
  STATL est un langage basé sur la spécification d'états et de transitions d'état. La *technique d'analyse de transition d'état* (STAT) a été présentée en 1992 dans [53]. Elle a été conçue pour décrire des séquences d'actions qui visent à compromettre la sécurité d'un système. Cette technique se destinait donc à une utilisation dans le cadre de la détection d'intrusions par signature. Une des motivations de la création de cette technique était de s'abstraire des détails d'une attaque. Les scénarios sont composés d'actions abstraites représentant ce qui se passe à bas niveau. De cette manière les créateurs de cette technique voulaient palier une faiblesse caractéristique des systèmes de détection à base de signature : leur incapacité à détecter des attaques nouvelles. En abstrayant les actions d'un scénario des détails de leur détection, il est possible de détecter ce scénario indépendamment des techniques utilisées pour réaliser chaque action le composant. De cette manière, il est possible de détecter les variantes d'un même scénario mais pas de nouveaux scénarios. Dans STAT, les scénarios sont représentés par des diagrammes de transition d'état. Les états représentent les propriétés du système en terme de sécurité et les transitions représentent les actions composant les scénarios. Plusieurs SDI utilisent cette modélisation mais des langages de signature différents ont été créés. Afin d'unifier l'utilisation de ces outils en ne spécifiant qu'une seule fois chaque scénario pour ensuite le compiler dans une forme utilisable par les SDI, le langage STATL a été créé (voir figure 2.8).
  Comme STAT, STATL modélise les scénarios par un ensemble d'états et de transitions. L'état du système est représenté par un ensemble de variables pertinentes pour le scénario décrit. Les transitions sont associées à une action dont la réalisation provoque un changement d'état du scénario. Trois types de transitions sont définies :
  - *consuming* : l'état dans lequel se trouve le scénario après transition invalide l'état précédent (transition $Timed\_out$ sur la figure 2.9).
  - *nonconsuming* : l'état dans lequel se trouve le scénario après transition n'invalide pas l'état précédent (transition SYN sur la figure 2.9).
  - *unwinding* : l'état dans lequel se trouve le scénario après transition a pu être atteint dans le passé (peut représenter l'effet d'une action annulant l'effet d'une autre action, transitions RST et ACK sur la figure 2.9).

  Un scénario déclaré dans le langage STATL a un nom, peut avoir des paramètres, des constantes et des variables. La signature du scénario est spécifié

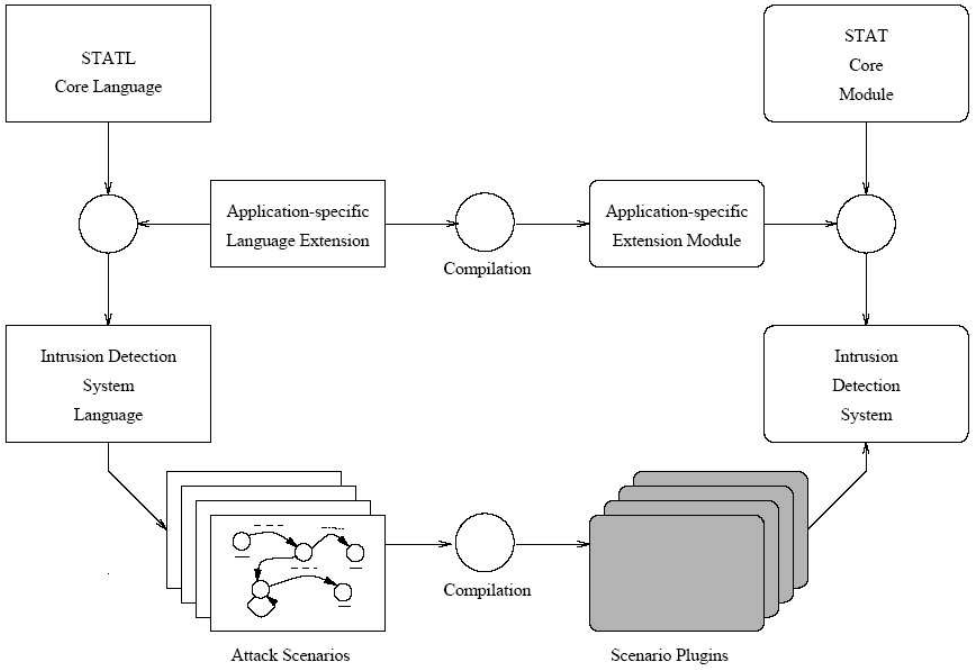

FIG. 2.8 – Le noyau STAT, STATL et leurs extensions

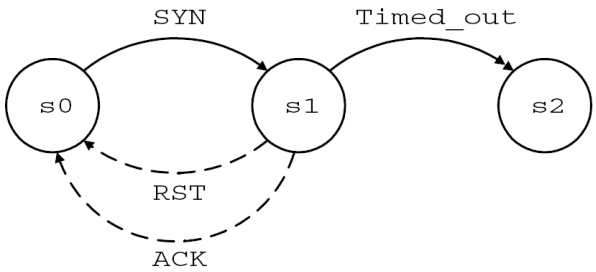

FIG. 2.9 – Exemple de scénario comportant les trois types de transition

par l'ensemble des états et des transitions (voir figure 2.10).
Un scénario est composé au minimum d'un état initial et d'un état final. L'état initial n'a pas de transition entrante et l'état final n'a pas de transition sortante. Le champ *FrontMatter* permet de déclarer des commentaires concernant le scénario et ses variables (locales à l'instance d'un scénario ou globales, c'est-à-dire partagées par toutes les instances d'un scénario) ainsi que ses constantes. La forme de ce champ est donnée dans la figure 2.11.
Les deux éléments les plus importants composant un scénario sont les états et les transitions. Un état a au minimum un nom, afin de pouvoir nommer les états quand une représentation graphique est utilisée, éventuellement une assertion à vérifier et un bloc de code à exécuter (voir figure 2.12).
L'assertion de l'état, c'est-à-dire la condition définissant les propriétés du système dans cet état, est vérifiée une fois que la transition permettant d'at-

```
Scenario ::=
   { use LibraryID {‘,’ LibraryID} ‘;’ }
   scenario ScenarioID
   [ScenarioParameters]
   ‘{’
       [FrontMatter]
       { State|Transition|NamedAction }
   ‘}’
   { FunctionDefinition }
```

FIG. 2.10 – Déclaration d'un scénario en STATL

```
Front Matter ::=
   { (Annotation|ConstDecl|VarDecl) }
ConstDecl ::=
   const Type ConstId ‘=’ InitialValue ‘;’
VarDecl ::=
   [global] Type VarId [‘=’ InitialValue] ‘;’
```

FIG. 2.11 – Déclaration du champ FrontMatter

teindre l'état a été vérifiée. Le bloc de code est quant à lui exécuté quand l'assertion de l'état a été vérifiée. L'assertion peut être par exemple la vérification d'une variable du scénario.
Une transition spécifie les deux états qu'elle connecte. Ces deux états peuvent être identiques, ceci permettant l'expression de la répétition d'une action par exemple. La figure 2.13 présente la forme d'une transition.
Le(les) élément(s) *EventSpec* de la déclaration d'une transition donne(nt) les événements à observer et les conditions à respecter pour valider la transition. La structure d'une expression *EventSpec* est spécifiée dans la figure 2.14. On peut noter qu'un événement peut être constitué de sous-événements, c'est-à-dire qu'il est possible d'exprimer qu'une transition est validée lorsqu'un ensemble d'actions organisées en arbre est observé.
La figure 2.15 présente un exemple de scénario déclaré en STATL. Ce scénario consiste à modifier le fichier .rhost d'un système afin de pouvoir ensuite se connecter via un rlogin sans avoir à donner de mot de passe. La modification du fichier .rhost se fait en utilisant le service ftp.
Le diagramme état-transition correspondant à ce scénario est présenté dans la figure 2.16. On notera que ce diagramme représente un scénario se basant sur la modification du fichier .rhost. Il représente donc une généralisation du scénario utilisant une connexion ftp pour modifier le fichier.
Le langage STATL ainsi que le noyau implantant le moteur de traitement des événements représentent l'architecture de développement STAT. Cette archi-

```
State ::=
   [initial]
   state StateId { Annotation }
   '{'
      [StateAssertion]
      [CodeBlock]
   '}'
```

FIG. 2.12 – Déclaration d'un état

```
Transition ::=
   transition TransitionID '(' StateId '->' StateId ')'
   (consuming   nonconsuming   unwinding)
   { Annotation }
   '{'
       ( '[' EventSpec ']'   ActionId )
       { Annotation } [':' Assertion]
       [CodeBlock]
   '}'
```

FIG. 2.13 – Déclaration d'une transition

tecture permet la création de SDI en faisant abstraction du domaine d'application (type de système d'exploitation surveillé, sonde réseau ou système, etc...), il est par exemple impossible de représenter en STATL les données relatives à une connexion TCP. Afin de pouvoir accéder aux données caractéristiques d'un domaine, le langage STATL ainsi que le moteur de traitement proposent des mécanismes d'extension. L'extension de STATL se fait en définissant de nouveaux types qui sont ensuite compilés dans une bibliothèque dynamique (fichiers *.so* pour les systèmes de type UNIX et *.dll* pour les systèmes windows). Cette bibliothèque est ensuite chargée par le moteur de traitement des événements pour interpréter les scénarios utilisant de nouveaux types. Les scénarios sont quant à eux compilés pour obtenir des librairies dynamiques, une bibliothèque par scénario. Ces bibliothèques sont appelées plugins de scénario. Plusieurs extensions ont été définies et implémentées : USTAT, NetSTAT, WebSTAT, LogSTAT, AlertSTAT, LinSTAT et WinSTAT. On remarquera que la maintenance d'un ensemble de SDI de la famille STAT est assez lourde car elle demande un ensemble d'opérations de compilation croissant avec le nombre d'extensions définies. De plus, ce langage a pour but de modéliser des scénarios entiers, la puissance de détection d'une sonde STAT est donc directement liée à l'exhaustivité de la bibliothèque de plans spécifiée.
L'environnement de détection d'intrusions développé autour de STAT et du langage STATL implante des mécanismes permettant de détecter des scénarios complexes mais nécessite beaucoup de travail dans son déploiement. En effet la compilation des scénarios pour obtenir des plugins spécifiques pour chaque

```
EventSpec ::=
   ( BasicEventSpec [SubEventSpec] )   TimerEvent
BasicEventSpec ::= EventType EventId
SubEventSpec ::= '[' EventSpec { ',' EventSpec } ']'
EventType ::= ANY   ApplEventType
   '(' ApplEventType { '|' ApplEventType } ')'
```

FIG. 2.14 – Déclaration d'un événement

plate-forme est une opération à renouveler à chaque fois que les scénarios sont modifiés.

Langages de modélisation d'actions

– CAML
Nous ne faisons pas ici bien sûr référence au célèbre langage CAML développé par l'INRIA, mais au langage développé par Steven Cheung, Ulf Lindqvist et Martin W. Fong. Ce langage ainsi que son utilisation sont exposés dans [17]. Ce langage a été élaboré dans le cadre du projet *Correlated Attack Modeling* (CAM). Le but est de proposer un langage qui puisse être utilisé par différents moteurs de corrélation. Ce langage permet de modéliser les étapes de scénarios d'intrusion. Une action est représentée par un module CAML et ses liens avec les autres modules sont exprimés par la spécification d'une pré-condition et d'une post-condition.
Le langage CAML est accompagné d'une bibliothèque de prédicats représentant le vocabulaire permettant de décrire les propriétés du système pour un modèle d'action.
Un module CAML est spécifié par trois sections :

  – *activity* : spécifie la liste d'événements à observer pour instancier un modèle d'action représenté par un module. Les événements CAML sont basés sur le format de données IDMEF.
  – *pre-condition* : condition devant être satisfaite par l'état du système pour valider l'action représentée par le module. Ce champ exprime aussi les conditions devant être vérifiées sur les événements déjà observés. Par exemple sur la figure 2.17, on peut voir que la pré-condition du module spécifie que les événements r_1 et r_2 doivent être tels que r_1 soit observé avant r_2.
  – *post-condition* : liste de prédicats ou d'événements inférés une fois que les champs *activity* et *pre-condition* sont vérifiés.

La figure 2.17 représente la modélisation de l'action consistant à exécuter localement du code qui permet à l'attaquant d'accéder à des données confidentielles. Le lecteur connaissant le langage IDMEF reconnaîtra les termes *Source*, *Node*, *Address*, etc. Plus de détails sur l'IDMEF sont donnés dans la section

```
use ustat;
scenario ftp_write
{
   int user;
   int pid;
   int inode;
   initial state s0 { }
   transition create_file (s0 -> s1)
   nonconsuming
   {
      [WRITE w] : (w.euid != 0) &&
                  (w.owner != w.ruid)
      { inode = w.inode; }
   }
   state s1 { }
   transition login (s1 -> s2)
   nonconsuming
   {
      [EXEC e] :
      match_name(e.objname, "login")
      {
         user = e.ruid;
         pid = e.pid;
      }
   }
   state s2 { }
   transition read_rhosts (s2 -> s3)
   consuming
   {
      [READ r] : (r.pid == pid) &&
                 (r.inode == inode)
   }
   state s3
   {
      {
         string username;
         userid2name(user, username);
         log("l'utilisateur %s a obtenu un accès local",username);
      }
   }
}
```

FIG. 2.15 – Exemple de scénario STATL

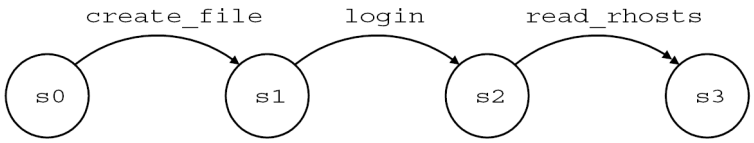

FIG. 2.16 – Diagramme état-transition du scénario de la figure 2.15

2.5.1) relatifs aux classes du modèle de données définies par l'IDMEF.
Il est important de noter qu'un module CAML ne correspond pas forcément à un événement détectable par les SDI. En effet, sur la figure 2.17 on peut remarquer que la post-condition du module est validée une fois que les deux événements r_1 et r_2 on été observés dans l'ordre r_1 puis r_2.
Un moteur de corrélation utilisant la description d'attaques en CAML a été implanté en utilisant le moteur d'inférence de P-BEST [64] et en traduisant les modèles CAML en règles P-BEST. Il est à noter que cette phase de traduction se fait manuellement. La mise en œuvre de ce moteur de corrélation a révélé des problèmes d'explosion combinatoire dans le moteur d'inférence P-BEST. Ce problème n'avait pas été mis en avant lors de précédentes utilisations du moteur P-BEST car les règles utilisées avaient peu de faits dans leurs antécédents. La traduction des modèles CAML en règles P-BEST donne des règles P-BEST avec des antécédents complexes. Les auteurs ne définissent pas de syntaxe ni de grammaire du langage. On constate que le langage repose largement sur la structure de l'IDMEF, ce qui impose son utilisation dans une architecture n'utilisant que ce format d'alerte.

– ATiKi

Dans [83], Steffan et Schumacher présentent un outil de découverte de scénario d'attaques. Ils proposent à la fois une représentation des scénarios par les réseaux de Pétri et une modélisation des actions composants les scénarios. Plus précisément, le modèle utilisé pour représenter un scénario est le modèle *'Attack Net'* présenté par Mc Dermott dans [68].
Le modèle ATiKi a deux éléments principaux :

– Conditions : elles décrivent les propriétés du système ainsi que les capacités de l'attaquant. Ces propriétés sont décrites de manière informelle par des prédicats qui sont évalués à faux ou vrai. Par exemple les prédicats *Unlimited failure logins are allowed* ou *valid password is known* ne comportent aucune variable.

– Transitions : elles sont définies par des pré-conditions et des post-conditions. Elles définissent comment un ensemble de pré-conditions permet d'atteindre un ensemble de post-conditions via l'exécution d'une action.

```
module Remote-Exec-Access-Violation-2-Data-Theft
(
   activity:
      r1: Event(
            Source(Node(Address(a: address)))
            Target(Node(Address(b: address)))
            Classification(origin == "vendor-specific"
                           name == "CAM-Remote-Exec"))
      r2: Event(
            Source(Node(Address(address == b)))
            Target(Node(Address(c: address)))
            Classification(origin == "vendor-specific"
                           name == "CAM-Access-Violation"))
   pre:
      StartsBefore(r1, r2)
   post:
      Event(
         starttime == r1.starttime
         endtime == r2.endtime
         Source(Node(Address(address == a)))
         Target(Node(Address(address == c)))
         Classification(
            origin == "vendor-specific"
            name == "CAM-Data-Theft"))
)
```

FIG. 2.17 – Accès à distance et vol de données modélisés en CAML

Brute-force guess password
Preconditions : [→ read access to /etc/passwd], [→ account with weak password]
Postconditions : [→ knowledge of password]
Contexts : [→ UNIX-like system], [→ Linux system] :
les systèmes Linux récents utilisent les mots de passe shadow, donc /etc/passwd ne contient pas les signatures des mots de passe.
Description : Un mot de passe peut être découvert s'il est inclus dans un espace de recherche suffisamment petit, tel que l'ensemble des combinaisons de lettres minuscules ou l'ensemble des mots anglais ou des noms anglais. Voir [→ account with weak password] pour plus de précisions sur les mots de passe faibles. Si la signature d'un mot de passe est connue, un attaquant peut deviner le mot de passe en ligne en générant la signature de chaque mot de passe dans l'espace de recherche et en comparant ces signatures avec la signature connue.

FIG. 2.18 – Transition liée à la détermination d'un mot de passe faible. Le symbole → représente un lien hypertexte ATiKi

Les conditions et transitions sont associées à des pages WiKi (voir [7]) pour permettre une navigation facile dans les graphes d'attaques. Une page WiKi est une page html qui peut être modifiée depuis un navigateur grâce à une syntaxe très simple. Ainsi les utilisateurs peuvent participer à la construction d'un site. Les graphes d'attaques sont générés automatiquement à partir des liens hypertextes présents dans les pré-conditions et les post-conditions des modèles. La figure 2.18 présente un modèle ATiKi pour une transition modélisant la découverte d'un mot de passe faible. On voit que les liens de corrélation entre les conditions sur les propriétés du système et de l'attaquant sont spécifiées explicitement par l'utilisation de liens hypertextes vers ces conditions.
Le champ **Contexts** de la figure 2.18 précise le contexte de l'action de détermination d'un mot de passe faible. Ce champ est destiné à faciliter la navigation dans les pages WiKi du système. Une fois l'ensemble des transitions et conditions écrites, une recherche est faite pour déterminer l'ensemble des graphes d'attaques. La figure 2.19 représente un exemple de graphe généré à partir d'un petit ensemble de transitions et de conditions. Le lecteur remarquera que cet outil est destiné à la découverte de graphes d'attaques mais ne se destine pas à la détection d'intrusions. Cependant même si les auteurs ne le mentionnent pas, il serait possible d'utiliser les graphes générés comme une base de modèles de scénario pour un SDI.

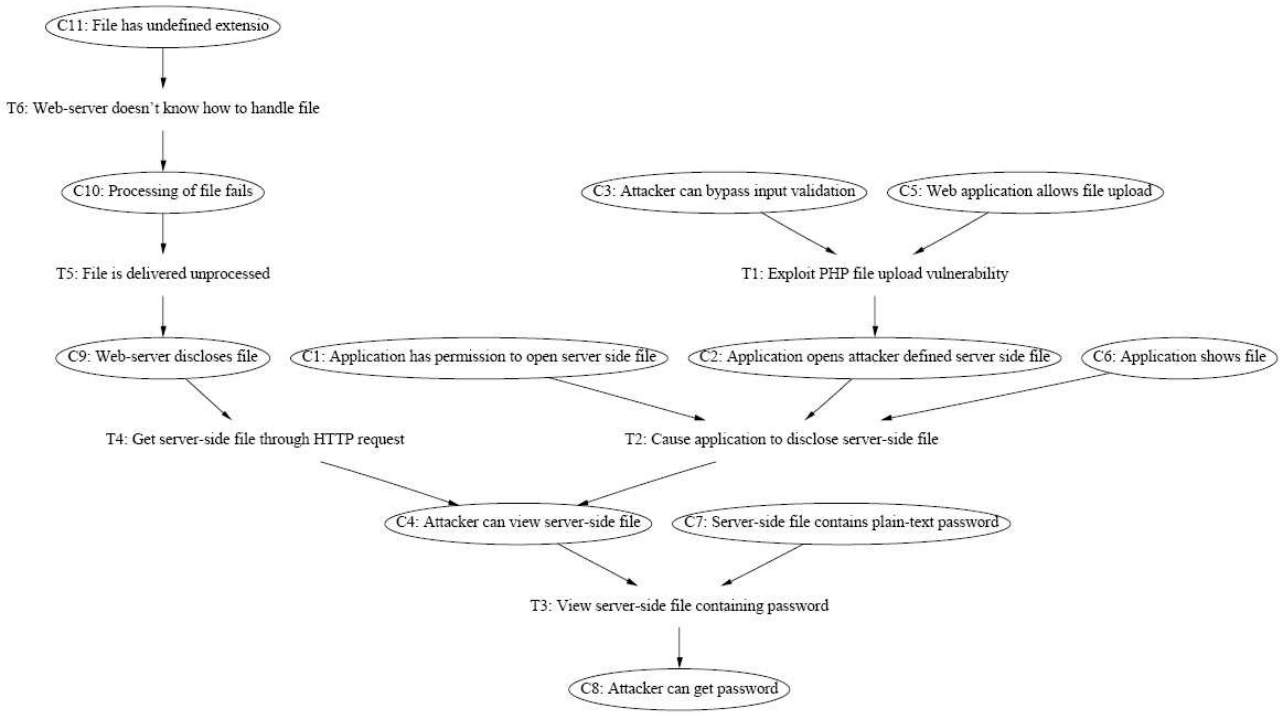

FIG. 2.19 – Exemple de graphe d'attaque découvert par le système ATiKi

– ADeLe

Le langage ADeLe (Attack Description Language)[69] a été développé dans le cadre du projet MIRADOR. Le but poursuivi par ce langage est commun à LAMBDA : permettre la spécification d'une base de données d'attaques afin de configurer un ensemble de SDI. A la différence de LAMBDA, qui adopte une approche déclarative, ADeLe est un langage procédural. Nous allons exposer la structure d'un modèle décrit en ADeLe tout en la comparant avec la structure d'un modèle LAMBDA étant donné que ces langages partagent plusieurs notions.
Une description d'attaque spécifiée en ADeLe est composée de trois parties :

– EXPLOIT : cette partie spécifie les conditions nécessaires pour exécuter l'attaque, la description de l'attaque elle-même (c.à.d le code de l'attaque ou ses différentes étapes) et le résultat de l'exécution de l'attaque. Cette partie est donc composée de trois parties : la pré-condition, le code et la post-condition. Contrairement à LAMBDA aucun langage n'est proposé pour exprimer les champs pré-condition et post-condition. Le cas de la partie réservée au code permet de spécifier la nature du langage utilisé pour l'exprimer. Il est ainsi possible d'inclure une fonction exprimée en C++ ou alors de spécifier l'attaque de manière informelle en précisant qu'il s'agit de texte. La spécification

du langage utilisé pour la description du code est utilisée pour sélectionner l'interpréteur ou le compilateur adéquat lorsque le fichier est lu. Les auteurs précisent qu'un langage de description d'attaque appelé EDL a été développé spécialement pour la description de la réalisation d'attaques.

- DETECTION : cette partie spécifie comment détecter l'attaque. Un langage de haut niveau est proposé pour exprimer la signature permettant de détecter un événement de bas niveau mais permet aussi d'exprimer un scénario complexe faisant intervenir des attaques connues. Cette partie se compose de trois sous-parties :

  - DETECT : cette partie se décompose en trois parties spécifiant respectivement les alertes et/ou les événements attendus lors de l'exécution de l'attaque, les contraintes temporelles entre ces alertes/événements et finalement les contraintes contextuelles entre ces alertes/événements. Afin d'exprimer en une seule expression comment les alertes/événements doivent s'enchaîner pour détecter un scénario complet d'attaque (l'attaque peut cependant se résumer à une seule action), un ensemble de 8 opérateurs est défini. Cet ensemble est constitué des opérateurs suivants : `;` (exprime une séquence d'événements), `Non_ordered{<events>}` (exprime un ensemble d'événements devant être observés), `One_among{<events>}` (exprime qu'un événement parmi un ensemble d'événements doit être utilisé pour une étape de l'attaque), `Subset_of{<events>}` (exprime qu'un sous-ensemble d'un ensemble d'événements doit être observé), `^` (exprime la répétition, par exemple E `^`n indique que l'événement E doit être observé n fois), `:=` (permet de nommer une occurrence particulière d'un événement), `WITHOUT` (exprime qu'un événement particulier ne doit pas être observé dans un intervalle d'événements spécifié) et finalement un opérateur de contrainte temporelle. Enfin la troisième partie de la section DETECT spécifie des contraintes sur les attributs des alertes/événements.

  - CONFIRM : cette section exprime ce qu'il faut vérifier sur le système surveillé afin de confirmer ou infirmer le succès de l'attaque. Un ensemble de fonctions sont fournies pour ce faire. Par exemple la fonction `Unreachable_Machine(<IP_address>)` retourne la valeur booléenne vraie si la machine spécifiée est indisponible.

  - REPORT : cette partie explicite comment remplir les champs de l'alerte générée lors de la détection d'une attaque exprimée en ADeLe.

- RESPONSE : dans cette partie sont exprimées les contre-mesures à exécuter lors de la détection de l'attaque. Plusieurs fonctions sont disponibles, par exemple la fonction `Kill_Process(target_ip,user_name,process_id|"ALL")` permet de terminer un processus sur une machine particulière. La

fonction `Script_Exec (script_name)` permet l'exécution d'un script.

Le champ RESPONSE nous parait ici mal spécifié car il ne permet de donner que des noms de contre-mesures existantes et de plus ne permet de décrire qu'une seule réponse. En effet pour une attaque donnée il peut exister plusieurs réponses, plus ou moins efficaces et dont la pertinence peut varier en fonction de l'attaque ou du système dans lequel l'attaque est observée. Il nous semble important de pouvoir proposer à l'administrateur plusieurs réponses pour une attaque, lui laissant ainsi le choix de lancer la contre-mesure la mieux adaptée. La même démarche peut être utilisée avec une réaction automatique. Dans ce cas, un outil de sélection a la charge de choisir la réaction la mieux adaptée à l'attaque.

– LAMBDA

Dans [24], les auteurs définissent un langage de description d'attaque déclaratif basé sur la logique. Ce langage a été développé dans le cadre du projet MIRADOR. Le but poursuivi est de pouvoir décrire une attaque indépendamment d'une technique d'attaque particulière et du type de machine concernée par l'attaque. A cette description sont ajoutées des informations sur la détection de l'attaque et sur la méthode à adopter pour vérifier si l'attaque est un succès. L'acronyme LAMBDA inclut le terme *attack* mais le langage permet de modéliser des actions malveillantes (qui violent la politique de sécurité) comme des actions suspectes (actions qui ne violent pas la politique de sécurité mais qui permettent l'exécution d'une action malveillante).
Un modèle d'attaque LAMBDA décrit l'attaque depuis plusieurs points de vue. Nous donnons tout d'abord une description informelle d'un modèle LAMBDA puis nous exposerons les différents langages utilisés dans un modèle. Un modèle LAMBDA décrit une attaque depuis deux points de vue : le point de vue de l'attaquant et le point de vue de la détection. Du point de vue de l'attaquant un modèle spécifie trois éléments :

– Un ensemble de conditions devant être satisfaites sur le système visé par l'attaque pour qu'elle puisse être réalisée.

– Les effets de l'exécution de l'attaque sur le système visé. Ces effets peuvent être une modification de l'état d'une machine (réalisation d'un déni de service, ouverture d'une connexion, etc...) ou peuvent concerner un utilisateur ou l'attaquant lui-même (par exemple un gain de connaissance ou l'obtention de privilèges réservés à l'administrateur).

– Le scénario correspondant à l'attaque. Ce scénario peut être composé de plusieurs actions pouvant elles-mêmes être éventuellement décomposées.

Du point de vue de la détection, un modèle LAMBDA spécifie comment

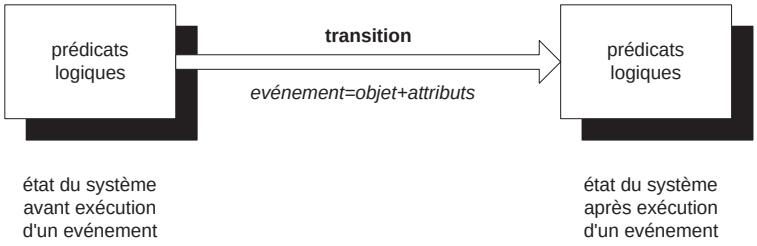

FIG. 2.20 – Modèle du système surveillé

détecter l'attaque. Cette description se décompose en trois parties :

– Les actions à mener sur le système surveillé pour détecter l'attaque. Il se peut qu'une attaque ne soit pas détectable, par exemple si l'action modélisée est exécutée sur l'ordinateur de l'attaquant. Ce champ a ensuite été redéfini et est maintenant utilisé pour décrire l'alerte associée à la détection de l'attaque. C'est cette dernière définition qui est retenue pour le reste de ce mémoire.

– La façon dont ces actions de détection doivent être combinées pour détecter l'attaque.

– Un ensemble d'actions de vérification permettant de quantifier l'impact de l'attaque sur le système.

LAMBDA est de plus un langage modulaire, ce qui permet de décrire une attaque à partir d'autre modèles d'attaque. La modularité d'un langage de modélisation d'attaque est un aspect important. Cela permet de faciliter la maintenance d'une base d'attaques et de décrire des attaques de base utilisées ensuite dans des scénarios plus complexes par exemple.
Nous allons maintenant exposer le langage LAMBDA ainsi que quelques exemples de modèles écris en LAMBDA. Le modèle adopté pour la représentation du système est présenté par la figure 2.20. La connaissance sur l'état du système est représentée en logique du premier ordre en utilisant des prédicats logiques. Trois langages sont utilisés dans un modèle décrit en LAMBDA :

– Description de l'état, langage L_1 : ce langage correspond à la logique des prédicats. Il sert à décrire l'état du système à travers les pré-conditions et post-conditions d'une attaque. Par exemple le prédicat *use_service(Address, netBios)* spécifie que la machine désignée par la variable *Address* utilise le service *netBios*. Ce prédicat peut être utilisé par exemple dans un modèle d'attaque dont l'exécution nécessite l'existence du service netBios sur la machine cible. Afin de combiner plusieurs prédicats dans une même expression, les opérateurs classiques $\wedge$, $\vee$ et $\neg$ peuvent être utilisés.
  Les effets d'une attaque ne se traduisent pas toujours par une modification de l'état d'une machine. L'exécution d'une attaque peut permettre à

un attaquant de récupérer des informations sur le système visé sans pour autant violer la politique de sécurité. Afin de représenter le gain de connaissance de l'attaquant, la modalité *knows* est définie. Par exemple le prédicat *knows(User,mounted_partition(Address,Partition))* spécifie que l'utilisateur *User* sait que la partition *Partition* de la machine à l'adresse *Address* est montée. Ce prédicat pourrait par exemple apparaître dans la post-condition du modèle LAMBDA de l'action *showmount*.

- Description des transitions, langage L_2 : dans la modélisation du système adoptée, les transitions sont associées à des événements. Le langage L_2 est basé sur les deux opérateurs $\wedge$ et $=$ et sur un ensemble de noms d'attributs. L'ensemble d'attributs considéré est $action, actor, date$. Afin de permettre la comparaison de données temporelles, L_2 contient aussi les opérateurs $<$ et $\leq$. La transition associée à un événement e peut s'exprimer en L_2 comme suit : $action(e) = a \wedge actor(e) = u \wedge date(e) = [t_1, t_2]$.
- Combinaison des événements, langage L_3 : le langage L_3 fournit les opérateurs permettant de combiner des événements. Ces opérateurs permettant de combiner deux événements e_1 et e_2 sont :
  - e_1 ; e_2 : composition séquentielle de e_1 puis e_2.
  - $e_1 \mid e_2$: exécution en parallèle de e_1 et e_2.
  - $\bar{e_1}[t_1, t_2]$: absence de e_1 dans le flux d'événements entre t_1 et t_2.
  - e_1 ? e_2 : représente le choix non déterministe entre e_1 et e_2.
  - e_1 & e_2 : exécution synchronisée de e_1 et e_2.

La description complète d'une attaque se fait en utilisant ces trois langages. Un modèle d'attaque décrit en LAMBDA sera représenté de la manière suivante :

attack : $attack\_name(arg_1, arg_2, ...)$
pre : $cond \in L_1$
post : $cond \in L_1$
scenario : $expr \in L_3$
where $cond \in L_2$
detection : $expr \in L_3$
where $cond \in L_2$
verification : $expr \in L_3$
where $cond \in L_2$

Dans un modèle LAMBDA les variables sont représentées par des termes commençant par une majuscule et les constantes par des termes commençant par une minuscule. Il est à noter que les variables utilisées dans un modèle sont locales à ce modèle. D'autre part, nous avons cité la modularité du langage, il est effectivement possible de référencer un modèle d'attaque comme étant une action associée à un événement, par exemple dans une clause **where**.
Voici un exemple d'attaque modélisé en LAMBDA. Cette attaque est composée de 6 étapes et permet d'exploiter une vulnérabilité de type mauvaise

configuration de la politique de sécurité pour obtenir un accès à une partition d'une machine. Les étapes composant l'attaque sont les suivantes :

1. **rpcinfo -p** *IP-cible*
   Cette commande permet à l'attaquant de savoir si le service *portmapper* et le démon NFS sont démarrés sur la machine cible.
2. **showmount -e** *IP-cible*
   L'attaquant obtient la liste des partitions du disque dur qui sont exportables.
3. **showmount -a** *IP-cible*
   L'attaquant obtient la liste des points de montage.
4. **finger @** *IP-cible*
   L'attaquant obtient l'identifiant d'un utilisateur de la machine et sait que le démon **finger** est démarré sur la machine cible.
5. **adduser** −**uid** *Userid Username*
   Cette étape est exécutée sur la machine de l'attaquant. Il ajoute un compte utilisateur sur sa machine en spécifiant les paramètres récupérés grâce aux étapes précédentes.
6. **mount -t***Target-partition* **mnt*
   Cette étape correspond à l'exécution de l'action violant la politique de sécurité. En effet l'attaquant obtient un accès illégal à la partition montée.

La figure 2.21 représente le modèle LAMBDA de l'attaque complète. Les quatre premières étapes font partie des étapes d'acquisition de connaissances sur la machine victime. Nous n'avons pas ici représenté les modèles des attaques élémentaires mais le scénario global. On remarque que le champ *detection* ne fait pas mention de l'étape exécutée localement sur la machine de l'attaquant. En effet, cette étape n'est pas détectable. Les champs *detection* et $verification$ sont destinés à spécifier les opérations à effectuer pour détecter et vérifier la réussite de l'attaque au niveau d'un SDI. Cette spécification séparée des tâches à accomplir pour détecter et vérifier l'attaque permet une plus grande flexibilité dans la spécification des attaques car ces opérations sont largement dépendantes de la plate-forme utilisée. Ainsi le champ $verification$ spécifie une fonction permettant de vérifier si quelqu'un a essayé de monter une partition depuis un ordinateur étranger au réseau surveillé. Le champ *detection* peut ainsi spécifier la signature de l'attaque.
Les informations logiques contenues dans les champs pré-condition et post-condition des modèles permettent d'envisager la génération automatique de scénarios complexes utilisant les modèles d'action élémentaires. Nous verrons dans le chapitre concernant la corrélation que c'est le langage LAMBDA qui a été choisi pour spécifier les actions élémentaires. Une fois ces actions spécifiées, il est possible de chercher les liens logiques existant entre ces modèles afin de corréler des alertes correspondant à l'instanciation de ces modèles (voir section 4).

attack : $NFS\_abuse(IP - cible)$

pre : $remote\_access(A, H) \wedge ip\_address(H, IP - cible)$
$\wedge use\_service(H, portmapper) \wedge use\_service(H, mountd)$
$\wedge exported\_partition(H, P) \wedge mounted\_partition(H, P)$
$\wedge connected\_user(U, H) \wedge userid(U, H, Userid)$
$\wedge use\_service(H, fingerd) \wedge root\_user(A, H_A)$
$\wedge connected\_user(A, H_A) \wedge owner(Directory, U)$

post : $can\_access(A, Directory)$

scenario : $((E_1; (E_2 \& E_3)) \& E_4 \& E_5); E_6$

where $action(E_1) =$**rpcinfo -p** *IP-cible*
$\wedge action(E_2) =$**showmount -e** *IP-cible*
$\wedge action(E_3) =$**showmount -a** *IP-cible*
$\wedge action(E_4) =$**finger @** *IP-cible*
$\wedge action(E_5) =$**adduser** $-$**uid** *Userid Username*
$\wedge action(E_6) =$**mount -t** *Target-partition* $\backslash$**mnt**
$\wedge actor(E_1) = A \wedge actor(E_2) = A$
$\wedge actor(E_3) = A \wedge actor(E_4) = A$
$\wedge actor(E_5) = A \wedge actor(E_6) = A$

detection : $((F_1; (F_2 \& F_3)) \& F_4); F_5$

where $action(F_1) = detect(E_1)$
$\wedge action(F_2) = detect(E_2)$
$\wedge action(F_3) = detect(E_3)$
$\wedge action(F_4) = detect(E_4)$
$\wedge action(F_5) = detect(E_6) \wedge date(F_5) = t$

verification : W_1

where $action(W_1) = foreign\_mount() \wedge date(W_1) = t'$
$\wedge t' \leq t$

FIG. 2.21 – Modèle de l'attaque exploitant le service NFS. Cette description est dérivée de la description de chaque action élémentaire composant le scénario.

C'est le langage LAMBDA que nous utilisons dans la section 4. Cependant nous verrons que nous ne l'utilisons pas pour modéliser des scénarios d'intrusion entiers comme sur la figure 2.21. En effet nous utilisons ce langage pour modéliser les actions élémentaires que l'attaquant peut exécuter pour attaquer un système informatique.

Nous avons présenté comment modéliser les actions de l'attaquant dans différent langages. La section suivante présente comment les informations relatives à la détection des actions exécutées par l'attaquant sont représentées.

2.5 Format des alertes

Quand un SDI détecte un événement, il génère un rapport sur cet événement contenant diverses informations. Ce rapport est appelé alerte de détection d'intrusions. Les informations contenues dans une alerte sont fonctions du type de SDI ayant généré l'alerte. En effet un SDI comportemental inclura les statistiques sur les variables mesurées ou attribuera une classe d'attaque si jamais un comportement suspect est observé. Un SDI observant le trafic réseau inclura des données sur les machines cible et source, sur le protocole utilisé, sur le nom de l'action détectée, etc... Un SDI observant les audits système inclura des données sur les processus à l'origine de l'événement ou visé par l'événement. Nous pouvons ajouter qu'une sonde consomme plus ou moins de ressources système pour effectuer ses traitements, le volume d'information que la sonde peut stocker dans une alerte peut être ainsi plus ou moins limité. De plus le format utilisé par chaque SDI est bien souvent spécifique au SDI utilisé. Parmi les formats les plus connus nous pouvons citer le format *Snort*.

Dans le contexte de la détection d'intrusions coopérative, étant donné que les alertes émises par les différentes sondes distribuées dans le système sont centralisées, il est nécessaire, pour des questions de facilité de déploiement, d'utiliser un même format d'alerte pour dialoguer entre les différentes entités de traitement des alertes. Un tel format d'alerte doit être suffisamment expressif pour pouvoir transmettre les informations générées par les différents types de SDI existants lors de la détection d'un événement. L'extensibilité d'un tel format est aussi une chose importante dans le cadre de la détection d'intrusions coopérative. En effet si l'on considère une chaîne de traitement d'alertes de détection d'intrusions, un module de traitement peut ajouter des informations à une alerte résultant du traitement d'autres alertes.

2.5.1 Introduction à l'IDMEF

Parmi les efforts de la communauté de la détection d'intrusions pour créer un format d'échange d'alertes, nous pouvons citer l'IDMEF (Intrusion Detection Message Exchange Format), développé par Hervé Debar et David Curry[27] dans le cadre des groupes de travail de l'IETF (Internet Engineering Task Force).

L'IDMEF a pour but de spécifier un format permettant de rapporter les informations liées à l'observation d'un événement suspect. L'utilisation première de l'IDMEF est le transport de données entre un SDI et une console de gestion des alertes par exemple. Mais l'IDMEF a été conçu pour être utilisé dans des situations

moins spécifiques. Par exemple il peut être utilisé pour le dialogue entre deux modules de traitement des alertes. Si nous nous plaçons dans le contexte de la détection d'intrusions coopérative, l'IDMEF permet de concevoir une architecture ou un SDI peut être enlevé ou ajouté facilement et permet d'introduire facilement de nouveaux modules de traitement des alertes.

Le modèle de données de l'IDMEF est orienté objet et est donc naturellement extensible en définissant de nouvelles classes dérivées ou en créant de nouvelles relations d'agrégation avec de nouvelles classes. Une fois le modèle étendu, une application qui était capable de traiter les alertes instanciées à partir du modèle non étendu sera capable de traiter les alertes du modèle étendu sans toutefois s'occuper des nouvelles données. Le modèle de données de l'IDMEF est spécifié par une DTD (Document Type Definition) XML. Cette spécification devrait évoluer vers un schéma XML.

Dans le cadre de notre implantation d'une architecture coopérative, nous avons utilisé l'IDMEF pour tous les échanges de données entre les différents modules de la chaîne de traitement des alertes. L'IDMEF est utilisé pour transmettre les alertes émises par les SDI. Il est aussi utilisé pour transmettre les alertes de fusion résultant de l'agrégation et de la fusion de plusieurs alertes et nous l'utilisons aussi pour les alertes de scénario et les alertes nécessitant l'exécution de contre-mesures (voir section 6).

2.5.2 Le modèle de données IDMEF

Dans cette section nous présentons le schéma de classes du modèle de données IDMEF 1.2. Il est à noter que l'implantation de ce modèle en XML ne permet pas de conserver les relations d'héritage entre les classes mères et filles dans le modèle. En effet XML ne permet d'exprimer que des relations d'agrégation ; les relations d'héritage sont donc remplacées par des relations d'agrégation.

La classe de base est la classe *IDMEF-Message* qui est ensuite dérivée en deux classes : *Alert* et *Heartbeat*. La classe *Heartbeat* est instanciée par des messages permettant de faire savoir qu'un SDI est en état de bon fonctionnement. La classe *Alert*, que nous allons détailler, est instanciée lorsqu'une alerte est émise. Nous ne nous détaillons pas ici la classe *Heartbeat* car nous nous intéressons qu'à la partie alerte de l'IDMEF. La figure 2.22 représente le schéma de classes constituant le modèle de données définissant une alerte.

Les classes possédant une relation d'agrégation avec la classe *Alert* et qui constituent les attributs de cette classe sont les suivantes :

– *Analyzer* : cette classe (voir figure 2.23) regroupe les informations relatives au SDI ayant généré l'alerte, les informations permettant d'identifier le SDI (adresse IP, numéro de processus, etc..) et permet également de savoir si l'alerte a déjà été traitée par un module. En effet une instance de la classe *Analyzer* peut avoir comme attribut une instance de cette même classe. Ainsi, il est possible d'ajouter une nouvelle instance de cette classe à chaque fois que l'alerte est traitée par un module pour retracer le chemin parcouru par l'alerte dans

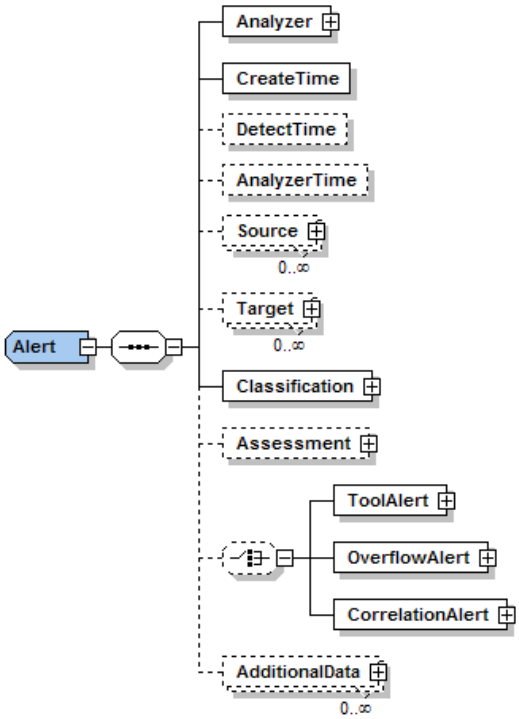

FIG. 2.22 – Structure de la classe *Alert*

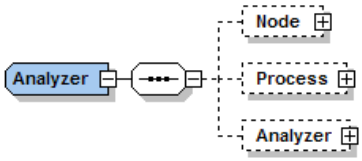

FIG. 2.23 – Structure de la classe *Analyzer*

une achitecture de traitement hierarchique. On remarquera qu'une instance de la classe alerte possède exactement une instance de la classe *Analyzer*.

– *CreateTime* : une instance de cette classe précise la date à laquelle l'alerte a été créée. Elle correspond à la date à laquelle les données incluses dans l'alerte ont été générées.

– *DetectTime* : une éventuelle instance de cette classe précise la date à laquelle l'événement à l'origine de la création de l'alerte a été détecté. Cette date est optionnelle et n'a pas forcément la même valeur que la date de création de l'alerte.

– *AnalyzerTime* : une éventuelle instance de cette classe précise la date courante de la sonde ayant émis l'alerte.

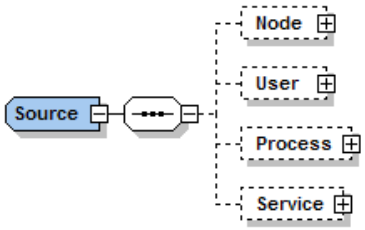

FIG. 2.24 – Structure de la classe *Source*

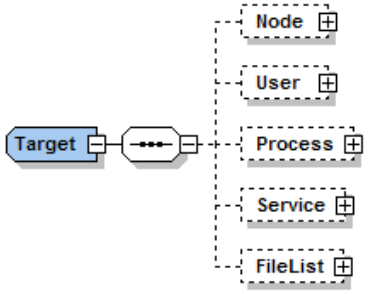

FIG. 2.25 – Structure de la classe *Target*

– *Source* : cette classe (voir figure 2.24) encapsule les informations relatives à l'entité à l'origine de l'événement ayant provoqué la génération de l'alerte. Une alerte peut ne pas avoir d'instance de cette classe mais peut aussi en posséder plusieurs. Par exemple, dans le cas d'une attaque distribuée, chaque instance de cette classe représenterait une machine utilisée pour attaquer. Une instance de cette classe précise éventuellement les informations sur l'utilisateur concerné, la machine concernée (type, adresse, etc..), le processus concerné et les services réseau concernés.

– *Target* : cette classe (voir figure 2.25) encapsule les informations relatives à la cible éventuellement visée par les événements à l'origine de l'alerte. Cette classe regroupe les même informations que la classe *Source* excepté la classe agrégé *Filelist* qui précise les fichiers éventuellement concernés par l'événement.

– *Classification* : une instance de cette classe précise le nom de l'événement associé à l'alerte. Ce nom est nécessaire pour les modules de traitement des alertes afin de classifier l'alerte et de déduire des propriétés liées à cette alerte. Les modules de traitement doivent ainsi connaître l'ensemble des noms utilisés par les SDI pour identifier les événements. Etant donné que les noms utilisés dépendent du SDI émettant une alerte, il est nécessaire d'utiliser un dictionnaire permettant d'établir des correspondances.

– *Assessment* : cette classe encapsule les informations relatives à la détection

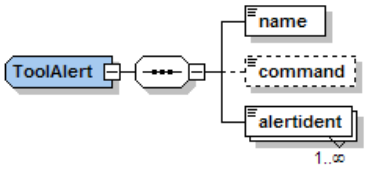

FIG. 2.26 – Structure de la classe *ToolAlert*

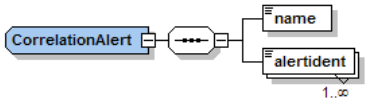

FIG. 2.27 – Structure de la classe *CorrelationAlert*

de l'événement à l'origine de l'alerte. Une éventuelle instance de cette classe précise l'impact de l'événement sur le système (si le SDI est capable de l'évaluer), précise le degré de confiance du SDI dans son diagnostic et donne les éventuelles actions exécutées lors de la détection.

– *ToolAlert* : cette classe (voir figure 2.26) encapsule les informations sur l'outil d'attaque éventuellement utilisé par l'attaquant. Elle permet de préciser un ensemble d'alertes toutes générées par l'occurrence d'événements créés par un outil d'attaque.

– *CorrelationAlert* : cette classe (voir figure 2.27) permet de préciser un ensemble d'identifiants d'alertes IDMEF référençant des alertes reliées entre elles. Elle permet aussi de préciser la technique qui a permi de regrouper ces identifiants.

– *OverflowAlert* : cette classe (voir figure 2.28) permet de préciser les données relatives à un événement du type dépassement de capacité (buffer overflow). Le SDI peut ainsi préciser le programme visé et la suite d'octets utilisée pour effectuer l'attaque.

– *AdditionalData* : cette classe (voir figure 2.29) permet d'inclure dans l'alerte des données non représentables dans le modèle IDMEF standard. Le modèle peut être alors étendu ou utiliser les types atomiques prédéfinis lorsqu'un faible volume de données doit être ajouté.

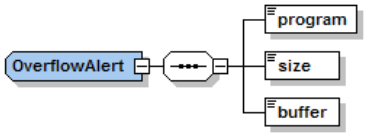

FIG. 2.28 – Structure de la classe *OverflowAlert*

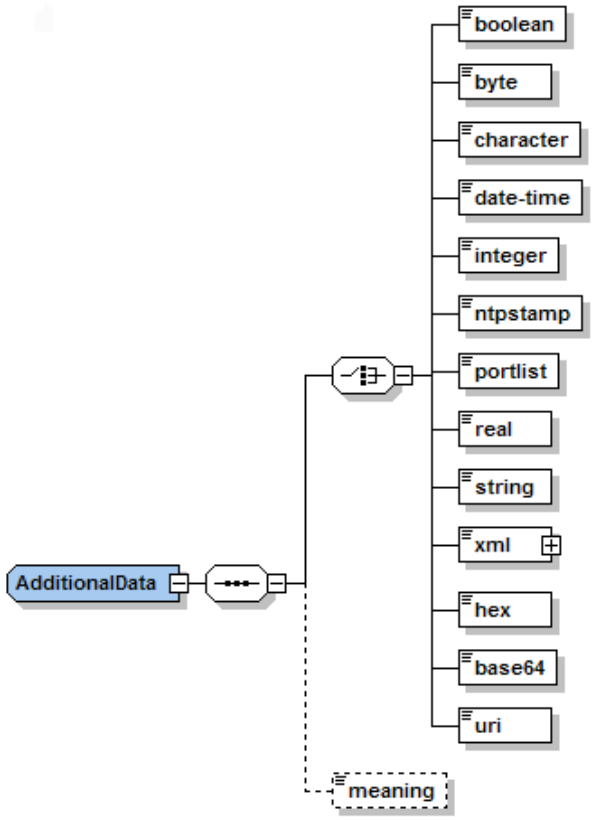

FIG. 2.29 – Structure de la classe *AdditionalData*

2.6 Conclusion

Nous avons présenté dans ce chapitre des approches coopératives de la détection d'intrusions ainsi que leur intérêt. Nous avons exhibé deux notions importantes dans le cadre de ces architectures : l'agrégation d'alertes et la corrélation d'alertes. Nous avons exposé les principaux langages de modélisation d'attaque permettant de représenter des scénarios d'intrusion ou des actions élémentaires exécutables par l'attaquant. Enfin nous nous sommes intéressés au formats d'alertes, plus particulièrement au modèle de données IDMEF. Dans la suite de ce mémoire, nous utilisons le langage LAMBDA pour modéliser les actions que peut exécuter l'attaquant. De plus, afin de faciliter la coopération de modules de traitement d'alertes, nous avons choisi d'utiliser le format IDMEF.

Chapitre 3

Agrégation et fusion d'alertes

Ce chapitre est consacré à la notion d'agrégation et de fusion d'alertes de détection d'intrusions. Les opérations d'agrégation et de fusion sont des processus importants dans le cadre de la détection d'intrusions coopérative car ce type d'architecture utilise un ensemble de SDI répartis dans le système à surveiller. Le fait qu'un sous-ensemble de ces SDI puisse détecter un même événement pose le problème de la redondance des alertes. Il devient nécessaire de trouver un moyen de regrouper les alertes ayant été produites lors de la détection d'un même événement. De plus, un même SDI peut générer plusieurs alertes pour la détection d'un seul événement. Une fois ces alertes regroupées, il faut alors fusionner les informations des alertes afin de produire une alerte synthétique.

Nous récapitulons les travaux de recherche se rapportant à la notion d'agrégation puis nous nous intéressons à l'approche qui a été développée dans le cadre de cette thèse. Ensuite nous présentons le processus de fusion d'alertes que nous avons développé.

3.1 La notion d'agrégation d'alertes

La notion d'agrégation d'alertes n'a pas été définie de manière précise à travers les différents articles abordant le problème des alertes redondantes et/ou similaires. Nous allons voir que le terme de "corrélation" est souvent employé pour désigner ce que nous appelons agrégation.

Avant de présenter les différentes approches se rapportant à la notion d'agrégation d'alertes, intéressons-nous à quelques exemples concrets de génération d'alertes similaires lorsqu'un type d'événement est détecté. Considérons un réseau surveillé par un ensemble de sondes :

- Un sous-ensemble de ces sondes peut détecter le même événement. Le nom associé à l'événement peut varier pour chaque SDI ainsi que la quantité d'information contenue dans chaque alerte. Les alertes générées ne sont pas identiques mais doivent être regroupées.
- Un attaquant peut lancer un balayage de port sur un ensemble d'adresses IP. Pour ce type d'action impliquant une unique adresse source et un ensemble

d'adresse cibles, il est intéressant de regrouper les alertes générées.
- Les alertes correspondant aux actions exécutées par l'attaquant consistant à forger de fausses adresses (spoofing) source mais visant un même ordinateur doivent être regroupées.
- Les attaques distribuées visant une machine particulière génèrent des alertes avec la même adresse cible et un ensemble d'adresses source. Une fois encore regrouper ces alertes permet de créer un seul groupe d'alertes pour une attaque distribuée.

Ces regroupements spécifiques aux événements détectés peuvent se faire en temps réel, mais d'autres regroupements peuvent se faire lors d'un traitement hors-ligne d'un ensemble d'alertes correspondant à une certaine période d'activité du réseau. Voici quelques exemples de regroupements pouvant être pertinents :
- les alertes concernant une certaine classe d'attaques sur une machine particulière ou un ensemble de machines. Par exemple, on peut regrouper les attaques portant sur les scripts cgi sur un ensemble de serveurs Web.
- les alertes relatives à une machine externe au réseau et une machine interne. Ceci peut permettre de vérifier si une machine connue pour avoir déjà attaqué une machine du réseau a continué ses attaques.

On remarquera que ces dernières opérations peuvent se faire facilement si les alertes sont stockées dans une base de données. Les groupes d'alertes correspondent dans ce cas à des vues.

3.1.1 L'algorithme AC (Agrégation et Corrélation)

Dans [29], Hervé Debar et Andreas Wespi présentent des algorithmes d'agrégation et de corrélation ainsi qu'une implantation sous la forme d'une console de détection d'intrusions. Cette interface est implémentée sur la base de la console Tivoli (TEC pour Tivoli Enterprise Console).

L'algorithme de corrélation désigne en fait deux types d'algorithmes de regroupement d'alertes. Le premier type de corrélation regroupe les alertes ayant entre elles un lien de conséquence (par exemple les alertes A_1 et A_2 sont regroupées car l'action associée à A_2 est une conséquence de l'action liée à A_1) et le deuxième type de corrélation consiste à regrouper les alertes relatives à la détection du même événement. L'article présente aussi un algorithme dit d'agrégation permettant de regrouper des alertes selon certains critères. Ce regroupement peut être comparé à une vue dans une base de données.

Dans ce chapitre nous nous intéressons à l'algorithme regroupant les alertes émises par la détection du même événement, appelées 'duplicatas' par les auteurs. Lorsque le moteur de corrélation reçoit une nouvelle alerte, il recherche dans les fichiers de configuration une définition de duplicata concernant le type d'événement associé à la nouvelle alerte. Un duplicata est défini par quatre termes :
- **Classe d'alerte initiale** : classe de l'alerte déjà reçue
- **Classe d'alerte dupliquée** : classe de l'alerte candidate
- **Liste d'attributs** : liste d'attributs devant être égaux pour associer les deux

alertes.

- **Niveau de gravité** : si une nouvelle alerte est associée à une alerte déjà reçue en utilisant la définition, le nouveau niveau de gravité de l'association des alertes est spécifié dans la définition.

On constate qu'une connaissance experte est nécessaire pour pouvoir associer des alertes relatives au même événement. Cet algorithme ne se base pas sur une mesure de la similarité entre deux alertes mais repose donc sur la spécification des alertes redondantes. On peut aussi voir l'ensemble des définitions des duplicatas comme un dictionnaire. Cependant la forme adoptée pour la définition de duplicata implique un certain ordre dans la réception des alertes. Cette définition permet en fait de spécifier les alertes que l'on risque de recevoir lors de la détection d'un certain événement et leur ordre de génération. Le fait de mettre à jour le niveau de gravité associé à un ensemble d'alertes permet de mettre à jour la connaissance sur l'attaque concernée. Ainsi, ce système permet par exemple d'exprimer qu'une attaque n'est dangereuse, ou effective, que lorsqu'un ensemble d'alertes est observé dans un certain ordre. L'exemple donné dans l'article concerne un serveur Web. Une attaque à base de script cgi peut être détectée grâce à la chaîne de caractères présente dans la requête, mais cette attaque n'est effective que si le serveur Web a accepté la requête. On voit donc ici que si l'on reçoit une alerte spécifiant que la requête est un succès après que l'on ait reçu une alerte spécifiant qu'une requête suspecte a été émise, on peut en conclure que l'attaque est dangereuse. Bien que la notion d'ordre soit présente dans la définition d'un duplicata, les classes d'alertes spécifiées dans une définition sont relatives à la détection du même événement. Faire référence à deux événements différents revient à spécifier une partie d'un scénario.

Nous avons précisé dans l'introduction de ce chapitre que l'algorithme d'agrégation présenté dans l'article fonctionne sur un principe équivalent aux vues d'une base de données. Cet algorithme permet de regrouper des alertes en imposant certaines conditions sur les attributs des alertes. Les auteurs appellent ces conditions sur les attributs des 'situations' en anglais, ce qui peut se traduire par des circonstances. Une circonstance est définie par quatre termes :

- **Classe d'alerte** : si une classe d'alerte est spécifiée, les alertes agrégées auront toutes cette classe.
- **Source** : spécifie l'adresse source commune à toutes les alertes agrégées.
- **Target** : spécifie l'adresse cible commune à toutes les alertes agrégées.
- **Niveau de gravité** : si le niveau de gravité de l'ensemble des alertes agrégées excède cette valeur, une alerte est générée.

Si la source, la cible ou la classe d'alerte ne sont pas spécifiées, alors ces différents critères ne constituent pas une contrainte d'agrégation.

Grâce au mécanisme d'agrégation il est ainsi possible de regrouper un ensemble d'alertes sur des critères correspondant à différentes situations d'attaque. Ceci permet par exemple la détection d'un ordinateur attaquant plusieurs serveurs web en spécifiant l'adresse source ainsi que la classe d'attaque mais pas l'adresse cible. Les attaques distribuées seront par exemple détectées en ne spécifiant pas d'adresse source mais la classe et la cible.

L'approche de l'agrégation d'alertes proposée dans [29] est intéressante dans la mesure où elle doit fournir un niveau de performance assez élevé étant donné la simplicité des règles d'agrégation. Cependant on peut ne pas adhérer à l'idée que l'administrateur doive spécifier ces règles pour tous les événements générant plusieurs alertes. En effet, aucun mécanisme d'agrégation par défaut n'est défini. De plus une règle ne concerne que deux classifications d'alerte, donc si un événement est référencé par de nombreuses classifications différentes par l'ensemble des SDI utilisés, il est nécessaire de définir une règle par couple de classifications.

3.1.2 Analyse de causes premières

Dans [59], Klauss Julisch propose d'agréger les alertes en les regroupant par "root cause" ou causes premières. La motivation principale de ce travail est que, selon Julisch, 90% des alertes de détection d'intrusions sont la manifestation de problèmes de configuration du système surveillé. Il propose donc de regrouper les alertes par cause afin d'identifier d'éventuels problèmes pouvant être résolus simplement. Il est important de remarquer que les travaux portant sur l'agrégation d'alertes similaires se concentrent sur le regroupement d'alertes relatives à la même attaque, la question des faux positifs n'étant pas prise en compte. L'approche de Julisch diffère de cette tendance en visant à agréger les faux positifs ayant comme origine le même problème de configuration. Une fois les alertes agrégées, il est alors plus simple pour l'administrateur système d'identifier le problème de configuration associé à chaque "cluster" afin de le régler.

L'auteur présente quelques problèmes de configuration pouvant entraîner la génération d'un grand nombre de faux positifs. Voici quelques exemples :

- Un serveur Real Audio : le trafic généré par un tel serveur ressemble à celui correspondant à une attaque de type 'TCP Hijacking'. Un SDI configuré pour détecter ce type d'attaque observant le trafic en provenance de ce serveur générera un nombre très élevé d'alertes.
- Un serveur HTTP avec une pile TCP/IP endommagée génère des paquets fragmentés. Des alertes peuvent alors être générées lorsque des utilisateurs font des requêtes au serveur Web.
- Des clients ftp Macintosh utilisent la commande SYST à chaque connexion à un serveur ftp. Cette commande peut générer des alertes car elle permet de reconnaître le type de machine faisant office de serveur ftp.

L'auteur propose de définir la notion de *dissimilarité* entre alertes. Etant donné deux alertes A_1 et A_2, un opérateur de *dissimilarité* d prenant en paramètre deux alertes retournera une valeur proche de zero si les deux alertes sont très similaires et une valeur d'autant plus élevée que les alertes sont différentes. Cette valeur peut aussi être vue comme la distance entre les deux alertes.

Avant de définir cet opérateur, l'auteur définit la manière dont les alertes sont modélisées. Une alerte est modélisée par un n-uplet sur le produit cartésien $dom(A_1)...$ $dom(A_n)$, où $A_1, ..., A_n$ est l'ensemble des attributs de l'alerte et $dom(A_i)$ est l'ensemble des valeurs que le paramètre A_i peut prendre.

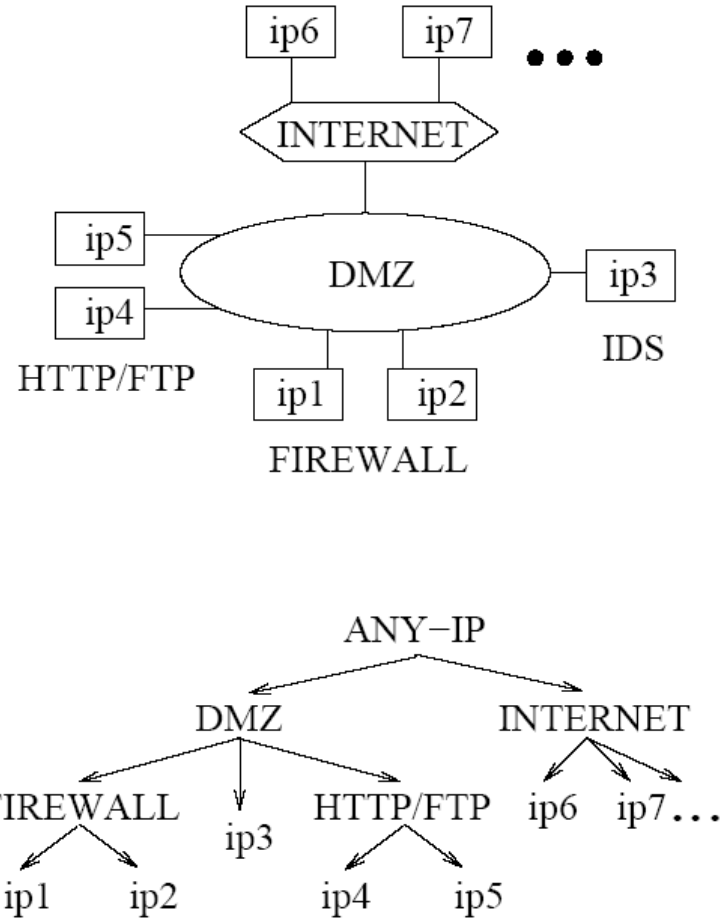

FIG. 3.1 – Exemple de taxonomie sur les adresses IP d'un réseau

Pour mesurer la distance séparant deux valeurs d'attributs, la notion de *valeur généralisée d'attribut* est définie. Une valeur généralisée d'un attribut A_i est un concept représentant un sous ensemble de $dom(A_i)$. Par exemple, la valeur généralisée Web_server pourrait représenter, pour un réseau donné, le sous-ensemble des adresses IP attribuées à des serveurs Web. Cette définition de valeur généralisée d'attribut permet la définition d'arbres de taxonomie pour chaque attribut de l'alerte. Julisch donne quelques exemples de taxonomie dont deux sont représentés sur les figures 3.1 et 3.2. La première figure représente la taxonomie sur les adresses IP d'un réseau représenté en haut de la figure. La deuxième figure représente une taxonomie possible sur l'ensemble des numéros de port. On peut noter que ces taxonomies sur les numéros de port et les adresses IP peuvent se rapprocher d'une hiérarchie de rôles et d'activités défini dans OrBAC[21].

La dissimilarité entre deux attributs appartenant au même domaine se calcule en cherchant le plus court chemin permettant de les relier dans l'arbre de taxonomie correspondant. Par exemple sur la figure 3.1, la dissimilarité entre les adresses $ip1$ et $ip7$ est de 5 et entre les adresses $ip1$ et $ip3$ elle vaut 3. La dissimilarité entre deux

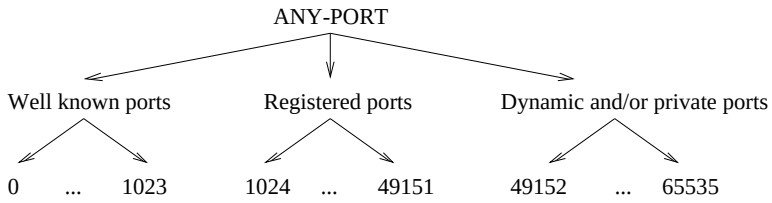

FIG. 3.2 – Exemple de taxonomie sur les numéros de port

alertes se calcule ensuite à partir des valeurs calculées entre les attributs. Cette valeur de dissimilarité entre deux alertes est la somme des valeurs de dissimilarité entre attributs. Parallèlement, la notion de valeur d'attribut généralisée est étendue à la notion d'alerte généralisée. Une alerte généralisée est l'ensemble des attributs généralisés d'une alerte. La mesure de dissimilarité entre une alerte et son alerte généralisée mesure le coût de la transformation de l'alerte en son alerte généralisée. Considérons deux alertes A_1 et A_2 ainsi qu'une alerte généralisée g et soit $d(.,.)$ l'opérateur de dissimilarité. Soient $d_1 = d(A_1, g)$ et $d_2 = d(A_2, g)$ les mesures de dissimilarité entre l'alerte généralisée et les deux alertes. Alors on peut dire que plus la somme $d_1 + d_2$ est faible, plus le coût de transformation de A_1 et A_2 vers g est faible et plus g est un modèle adéquat des deux alertes.

Le problème de l'agrégation d'alertes peut ainsi être traité après avoir défini cet opérateur de dissimilarité. Il reste cependant à définir la notion de dissimilarité entre un ensemble d'alertes et une alerte généralisée. Pour ce faire Julisch définit un opérateur de dissimilarité moyen $\bar{d}$ qui est la moyenne des valeurs de dissimilarité d'un ensemble d'alertes et d'une alerte généralisée.

Ces opérateurs étant définis, Julisch définit le problème de l'agrégation d'alertes comme suit : soit un ensemble d'alertes $\mathcal{L}$ et un entier *taille_min* donnant la taille minimale de l'ensemble d'alertes recherché. Il s'agit de trouver un sous-ensemble d'alertes $\mathcal{C}$ dans $\mathcal{L}$ contenant un nombre d'alertes supérieur ou égal à *taille_min* et une alerte généralisée g minimisant $\bar{d}(\mathcal{C}, g)$. Afin de traiter un ensemble d'alertes, il est nécessaire de réitérer l'opération d'agrégation sur l'ensemble $\mathcal{L}$ privé des alertes agrégées lors de la dernière itération.

Ce problème étant NP-complet, l'auteur propose de le résoudre grâce à une heuristique permettant d'obtenir des ensembles d'alertes $\mathcal{C}$ respectant la contrainte $\mathcal{C} \geq taille\_min$ mais ne minimisant pas forcément $\bar{d}(\mathcal{C}, g)$.

Le problème de cette approche, comme d'autres approches de regroupement telle que la méthode des k plus proches voisins, est qu'il est nécessaire de spécifier le paramètre *taille_min*. L'auteur ne fournit pas de méthode ou d'heuristique permettant de déterminer une "bonne" valeur de *taille_min*, la détermination de ce paramètre restant à la charge de l'administrateur. L'intêret de cette approche est de diminuer le nombre de faux positifs. On peut souligner qu'elle est la seule à notre connaissance qui se propose d'identifier les origines de certains faux positifs afin de diminuer le nombre de faux positifs à traiter.

3.1.3 Corrélation probabiliste d'alertes

Dans [86], Alfonso Valdes et Keith Skinner proposent de corréler en temps réel des alertes provenant de SDI distribués. Ils définissent la notion de corrélation comme une relation de similarité entre alertes. Cette relation de similarité est définie par un opérateur de similarité prenant en paramètre deux alertes et retournant un réel dans l'intervalle $[0, 1]$.

Les auteurs ne donnent pas de modèle d'alerte mais précisent que leur module d'agrégation est intégré à l'architecture d'EMERALD et donc utilise les formats et modèles définis pour cette architecture. Pour chaque attribut d'alerte, ils définissent une fonction de similarité retournant un réel dans l'intervalle $[0, 1]$ étant donné deux attributs. Les valeurs de similarité calculées entre une alerte candidate (Y) et un ensemble d'alertes (X, appelé meta-alerte) sont ensuite pondérées par des valeurs de similarité attendues puis sommées :

$$SIM(X, Y) = \frac{\sum_j E_j SIM(X_j, Y_j)}{\sum_j E_j}$$

Les valeurs de similarité attendues (E_j) permettent d'abandonner la comparaison si jamais une valeur de similarité est en dessous de la valeur attendue. La figure 3.3 représente la matrice de similarité utilisée dans le calcul de similarité pour l'attribut spécifiant la classe de l'événement associé à l'alerte. Les auteurs ne précisent pas comment ils construisent cette matrice ni l'approche adoptée pour déterminer les classes d'événements utilisées.

Pour le calcul de similarité entre deux adresses IP, les auteurs proposent de comparer l'appartenance de ces adresses à des sous-réseaux. Deux adresses appartenant au même sous-réseau sont plus similaires que deux adresses appartenant à des sous-réseaux différents. Le calcul de similarité sur deux listes de ports est effectué en comparant le sous-ensemble commun aux deux listes.

Peu de détails sont donnés sur la méthode utilisée pour fusionner les informations d'un ensemble d'alertes. L'exemple donné dans l'article concerne le cas simple de la fusion de listes de ports. Mais les auteurs argumentent que la plupart des opérations de fusion se résument à la fusion de listes d'attributs.

Cette approche de l'agrégation et de la fusion d'alertes se rapproche de la méthode que nous avons développée. Cependant le modèle d'alerte utilisé ici est plus simple et contient moins d'information que l'IDMEF. D'autre part les valeurs de similarité attendues ne sont pas définies en fonction de la classe de l'événement associé à l'alerte. Cela peut engendrer le calcul de valeurs de similarité biaisées car la signification et l'importance de certains attributs d'une alerte est fonction de la classe de l'événement qui lui est associé.

3.1.4 Définition d'un prédicat logique modélisant la similarité

Dans [18], Frédéric Cuppens définit un prédicat logique $sim\_alert(Alertid_1, Alertid_2)$ retournant vrai si deux alertes sont similaires. Ce prédicat utilise un autre

	INVALID	PRIVILEGE_VIOLATION	USER_SUBVERSION	DENIAL_OF_SERVICE	PROBE	ACCESS_VIOLATION	INTEGRITY VIOLATION	SYSTEM_ENV_CORRUPTION	USER_ENV_CORRUPTION	ASSET_DISTRESS	SUSPICIOUS_USAGE	CONNECTION_VIOLATION	BINARY_SUBVERSION	ACTION_LOGGED
INVALID	1	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.6
PRIVILEGE_VIOLATION	0.3	1	0.6	0.3	0.6	0.6	0.6	0.6	0.4	0.3	0.4	0.1	0.5	0.6
USER_SUBVERSION	0.3	0.6	1	0.3	0.6	0.5	0.5	0.4	0.6	0.3	0.4	0.1	0.5	0.6
DENIAL_OF_SERVICE	0.3	0.3	0.3	1	0.6	0.3	0.3	0.4	0.3	0.5	0.4	0.1	0.5	0.6
PROBE	0.3	0.2	0.2	0.3	1	0.7	0.3	0.3	0.3	0.3	0.4	0.8	0.3	0.6
ACCESS_VIOLATION	0.3	0.6	0.3	0.5	0.6	1	0.6	0.6	0.3	0.3	0.4	0.1	0.5	0.6
INTEGRITY VIOLATION	0.3	0.5	0.3	0.5	0.6	0.8	1	0.6	0.5	0.3	0.4	0.1	0.5	0.6
SYSTEM_ENV_CORRUPTION	0.3	0.5	0.3	0.5	0.6	0.6	0.6	1	0.6	0.3	0.4	0.1	0.5	0.6
USER_ENV_CORRUPTION	0.3	0.5	0.5	0.3	0.6	0.6	0.6	0.6	1	0.3	0.4	0.1	0.5	0.6
ASSET_DISTRESS	0.3	0.3	0.3	0.6	0.3	0.3	0.3	0.3	0.3	1	0.4	0.4	0.3	0.6
SUSPICIOUS_USAGE	0.3	0.3	0.5	0.3	0.5	0.6	0.5	0.6	0.5	0.3	1	0.1	0.3	0.6
CONNECTION_VIOLATION	0.3	0.1	0.1	0.3	0.8	0.3	0.3	0.3	0.3	0.5	0.4	1	0.3	0.6
BINARY_SUBVERSION	0.3	0.3	0.3	0.3	0.3	0.6	0.6	0.6	0.5	0.3	0.4	0.1	1	0.6
ACTION_LOGGED	0.3	0.3	0.3	0.3	0.6	0.5	0.3	0.3	0.3	0.3	0.4	0.3	0.3	1

FIG. 3.3 – Matrice de similarité entre classes d'événements

prédicat $sim\_entity(Type\_entity, Entity_1, Entity_2)$ retournant vrai si deux entités du même type sont similaires. Ici le nom d'entité désigne les instances de classes du modèle de données IDMEF.

Afin d'évaluer ces prédicats, un ensemble de règles expertes doivent être définies afin de pouvoir comparer les attributs des alertes. Ces règles ont été définies pour les attributs suivants : classification, date, source, cible.

Afin de tester si deux classifications sont similaires, un une bibliothèque de correspondances a été définie. En effet les SDI ne génèrent pas nécessairement la même classification pour la détection d'un même événement. Dans ce cas, un dictionnaire établissant les correspondances entre les classifications est utilisé. La comparaison des dates de détection se fait en vérifiant que l'écart entre deux dates ne dépasse pas un certain délai spécifique à au type d'événement détecté. Ce délai est déterminé de manière expérimentale en rejouant certaines attaques sur un réseau de test. La comparaison des données sur les machines source et cible implique la définition d'une table de correspondance entre les adresses IP et les noms des machines du réseau surveillé. D'autre part, comme nous l'avons dit au début de cette partie, le type d'événement détecté conditionne les comparaisons faites entre les données source et cible.

Cette approche est intéressante dans la mesure où la comparaison des attributs de deux alertes est fonction de leur classification. Cependant la mesure de similarité étant définie par un prédicat logique, il n'est pas possible de quantifier cette similarité. Par exemple considérons deux alertes A_1 et A_2 et une alerte candidate A_3, A_1 et A_2 n'étant pas similaires. Si l'alerte candidate est à la fois similaire à A_1 et A_2, il apparaît un problème pour associer cette alerte à A_1 ou A_2 car nous ne sommes pas capables de quantifier cette similarité.

3.2 Nouvelle approche pour l'agrégation d'alertes

Nous avons vu dans la partie précédente, à travers les différents travaux de recherche présentés, que l'agrégation d'alertes similaires est une nécessité dans le contexte de la détection d'intrusions coopérative. Les différentes méthodes évoquées s'appuient soit sur des règles d'agrégation, ce qui nécessite une connaissance experte, soit sur la définition d'une notion de similarité entre alertes.

L'approche que nous présentons s'appuie sur la définition d'un opérateur de similarité permettant d'évaluer, à travers le calcul d'un réel appartenant à $[0, 1]$ étant données deux alertes, dans quelle mesure ces deux alertes se rapportent à la détection du même événement.

Le fait d'essayer de quantifier l'association de deux alertes à un même événement n'est pas une approche rigoureuse. En effet, comme nous l'avons montré dans les exemples de cas où le regroupement d'alertes est nécessaire (outre la gestion des alertes redondantes), l'événement en commun à toutes les alertes regroupées peut ne pas être l'événement associé à chaque alerte mais un événement de plus haut niveau. Par exemple, lorsque nous avons pris le cas du regroupement d'alertes relatives à un balayage d'un ensemble d'adresses IP, l'événement associé à chaque alerte met

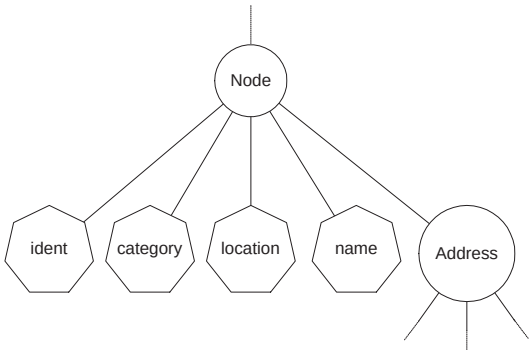

FIG. 3.4 – Représentation générique d'une alerte sous la forme d'un arbre

en jeu seulement deux machines mais l'événement associé à l'ensemble des alertes regroupées concerne une machine source et un ensemble de machines cibles. Ce problème pourrait être traité par la corrélation, mais dans ce cas il est plus simple de traiter par la fusion. En effet, nous pourrions corréler l'ensemble des alertes relatives au balayage d'adresses IP exécuté depuis une même machine, mais il est moins couteux en calcul de produire une seule alerte contenant la liste des adresses balayées.

Avant de présenter l'opérateur de similarité que nous avons défini, nous commençons par définir la modélisation des alertes que nous utilisons. Nous exposons ensuite comment nous mesurons la similarité entre les attributs de deux alertes et comment nous définissons, à partir de ces mesures de similarité entre attributs, un opérateur de similarité entre alertes. Enfin, nous présentons les algorithmes d'agrégation que nous utilisons.

3.2.1 Modélisation des alertes

Avant de présenter l'opérateur de similarité que nous avons défini, nous présentons le modèle adopté pour représenter les alertes. Nous avons choisi le modèle de données orienté objet défini dans l'IDMEF.

Un modèle plus générique d'une alerte peut être représenté par une hiérarchie d'attributs, cette hiérarchie d'attributs étant représentée par un arbre. La figure 3.4 représente un exemple d'une partie d'une hiérarchie d'attributs. Chaque noeud dans cet arbre représente un concept, par exemple la notion de *Node* dans l'IDMEF se rapporte à la localisation physique d'un matériel dans un réseau ; cela peut être un routeur, un firewall, un serveur Web, etc...

Ce modèle peut être vu comme une généralisation du modèle adopté dans [59]. Le modèle adopté par Julisch peut être représenté par un arbre avec une racine et autant de feuilles qu'il y a d'attributs à représenter (figure 3.5).

Dans [86], les auteurs attribuent une valeur de similarité attendue pour chaque calcul de similarité entre attributs de deux alertes. La pondération du résultat de la comparaison entre attributs est un mécanisme important car l'information détenue

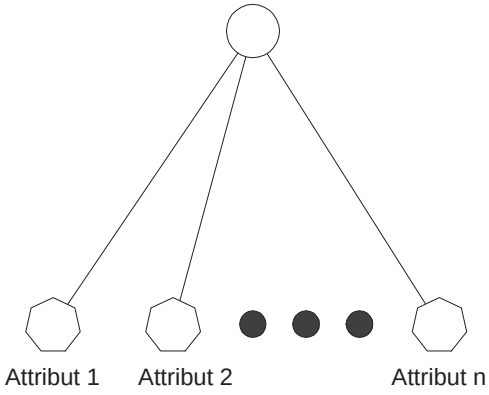

FIG. 3.5 – Représentation simple d'une alerte sous la forme d'un arbre

par chaque attribut est plus ou moins pertinente en fonction de l'événement associé à l'alerte. Nous avons présenté quelques exemples dans la sous-section 3.1 où les attributs n'ont pas tous la même importance. Le fait d'adopter une représentation hiérarchique des attributs permet de pondérer un ensemble de valeurs de similarité en agissant sur le poids de l'attribut parent. Par exemple il peut être souhaitable pour une opération d'agrégation particulière de ne pas prendre en compte l'ensemble des informations concernant la source dans les alertes traitées. Dans ce cas il suffit de modifier le poids que nous associons à la classe *Source* dans le modèle de données IDMEF pour supprimer l'influence de la valeur de similarité calculée entre deux instances de cette classe.

Afin d'associer de manière automatique les poids au calcul de similarité entre deux alertes, nous définissons plusieurs ensembles. Tout d'abord nous définissons l'ensemble des événements $\mathcal{E}_{vt}$ détectables par les SDI. Cet ensemble contient le nom de tous les événements qui peuvent être détectés. Étant donné que chaque SDI n'utilise pas forcément le même nom pour un même événement, il est nécessaire de définir un dictionnaire permettant d'associer une alerte à un élément de $\mathcal{E}_{vt}$.

Définition 1 *Ensemble fini des événements détectables $\mathcal{E}_{vt}$:*

L'ensemble $\mathcal{E}_{vt}$ contient les noms des événements détectables par l'ensemble des SDI utilisés. Chaque événement détectable n'est référencé qu'une seule fois dans cet ensemble.

$\mathcal{E}_{vt}$ contient au moins l'événement *unknown* qui est attaché aux alertes dont l'événement associé est inconnu.

Nous définissons ensuite l'ensemble des poids $\mathcal{P}_{vt}$. Un élément de $\mathcal{P}_{vt}$ est constitué d'un ensemble de réels. Le nombre de réels d'un élément de $\mathcal{P}_{vt}$ est égal au nombre de classes et d'attributs définis dans le modèle de données de l'IDMEF, c'est-à-dire qu'un élément de $\mathcal{P}_{vt}$ définit les poids sur tous les éléments d'une alerte IDMEF. D'après la dernière définition du format IDMEF, une instance de la classe $Classification$ et une seule doit exister dans une alerte. Nous utilisons cette unique

instance de la classe $Classification$, qui contient les informations permettant d'identifier l'événement associé à l'alerte, pour associer un élément de $\mathcal{E}_{vt}$ à l'alerte.

Définition 2 *Ensemble des poids $\mathcal{P}_{vt}$:*
L'ensemble $\mathcal{P}_{vt}$ contient les poids définis pour tous les éléments de $\mathcal{E}_{vt}$. Un élément de $\mathcal{P}_{vt}$ est de la forme $p_1, p_2, ..., p_n$, n étant le nombre d'attributs et de classes du modèle IDMEF.

Nous avons dit que pour chaque classe du modèle nous définissons un poids. Cependant certaines classes sont utilisées comme attributs d'autres classes. Par exemple la classe $Node$ est un attribut de la classe $Source$ mais aussi de la classe $Target$. Dans ce cas deux poids sont définis pour chaque classe dans le modèle et leur identification se fait par le chemin parcouru dans la hiérarchie de classes pour les atteindre. Concrètement on définit un poids pour la classe *IDMEF-Message.Alert.Source.Node* et un autre poids pour *IDMEF-Message.Alert.Target.Node*. En revanche, le même poids est utilisé pour toutes les instances d'une même classe.

Afin de sélectionner l'élément de $\mathcal{P}_{vt}$ correspondant à un événement de E_{vt}, nous définissons la fonction $p : \mathcal{E}_{vt} \mapsto \mathcal{P}_{vt}$. Cette fonction retourne l'ensemble des réels à utiliser étant donné un événement. Ainsi, nous pouvons pondérer l'influence des valeurs de similarité calculées entre les attributs de deux alertes en fonction des caractéristiques de l'événement associé à l'alerte (voir la section 3.1 pour quelques exemples d'événements).

3.2.2 Similarité entre alertes

Nous allons maintenant définir comment nous comparons deux alertes, c'est-à-dire comment nous calculons la valeur de similarité entre deux alertes. Afin de calculer cette valeur nous devons comparer l'ensemble des attributs des deux alertes. Cette comparaison se fait grâce à des fonctions de similarité définies pour chaque attribut. Nous définissons l'ensemble $\mathcal{F}_s$ contenant l'ensemble des fonctions de similarité.

Définition 3 *Ensemble des fonctions de similarité $\mathcal{F}_s$:*
L'ensemble $\mathcal{F}_s$ contient les fonctions de similarité nécessaires pour calculer une valeur de similarité étant données deux instances d'attributs du même type de base.

Tout comme pour les poids, afin de sélectionner la fonction de similarité correspondant à une instance d'un type de base, nous définissons la fonction $f_s : \mathcal{T} \mapsto \mathcal{F}_s$, $\mathcal{T}$ étant l'ensemble des attributs définis dans l'IDMEF. Tout comme les poids, la fonction de similarité correspondant à un attribut de classe ne peut être identifié seulement par le type de l'attribut. L'identification se fait donc encore une fois par le chemin parcouru dans la hiérarchie de classes.

Contrairement aux poids définis sur l'ensemble des attributs et sur l'ensemble des classes composant le modèle de données IDMEF, les fonction de similarité ne sont définies que sur les attributs. En effet, nous allons voir comment calculer la valeur de similarité entre deux instances de la même classe IDMEF à partir de l'ensemble des valeurs de similarité de leurs attributs.

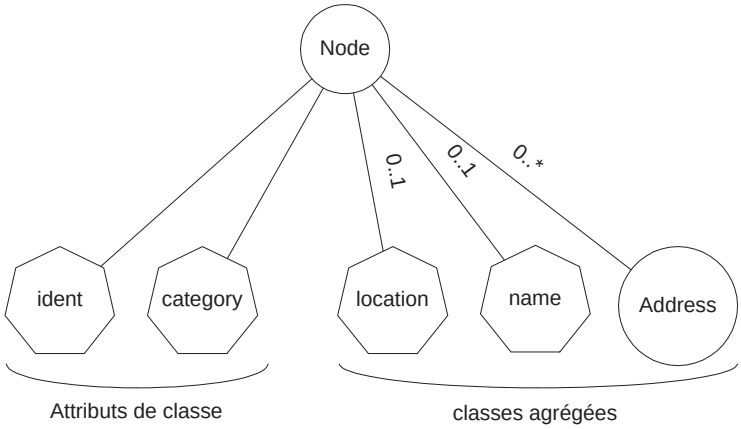

FIG. 3.6 – Arbre xml de de la classe Node

Prenons l'exemple de la classe *Node*. Cette classe encapsule les informations permettant d'identifier une machine du réseau ou tout autre matériel connecté (routeurs, switches, etc...). La figure 3.6 représente l'arbre XML de cette classe.

Les heptagones représentent des classes correspondant à des types de base, l'attribut *ident* étant une chaîne de caractères et *category* étant une énumération, et les cercles représentent des classes. Afin de comparer deux instances de la classe *Node*, il est donc nécessaire de comparer les deux instances de l'attribut *ident*, de l'attribut *category* puis de l'ensemble des classes agrégées. La comparaison des deux instances de l'attribut *ident* qui correspond à un type de base, fait intervenir la fonction de similarité associée à cet attribut. La fonction de similarité est définie par le chemin parcouru dans la hiérarchie de classes pour atteindre l'instance d'*ident*, tout comme les poids associés. La comparaison de deux instances de la classe *Address*, correspondant à un type abstrait, ne fait pas appel à une fonction de similarité, mais se calcule à partir des valeurs de similarité obtenues en comparant les instances d'attributs des deux instances de la classe *Address*.

Cependant le nombre d'instances des classes *location*, *name* et *Address* peut varier car les relations d'agrégation entre ces classes et la classe Node sont du type 0...1 (attribut optionnel) et 0...$*$ (nombre d'instances de l'attribut illimité).

Ceci étant, plusieurs problèmes apparaissent :

- Le nombre d'instances de certains attributs étant variable, il est nécessaire de définir comment comparer deux ensembles d'instances du même attribut.

- Une fois l'ensemble des valeurs de similarité calculées entre les différentes instances des attributs de deux instances C_1 et C_2 de la même classe C, il est nécessaire de définir comment agréger ces valeurs en utilisant les poids associés afin de calculer la valeur de similarité finale entre C_1 et C_2.

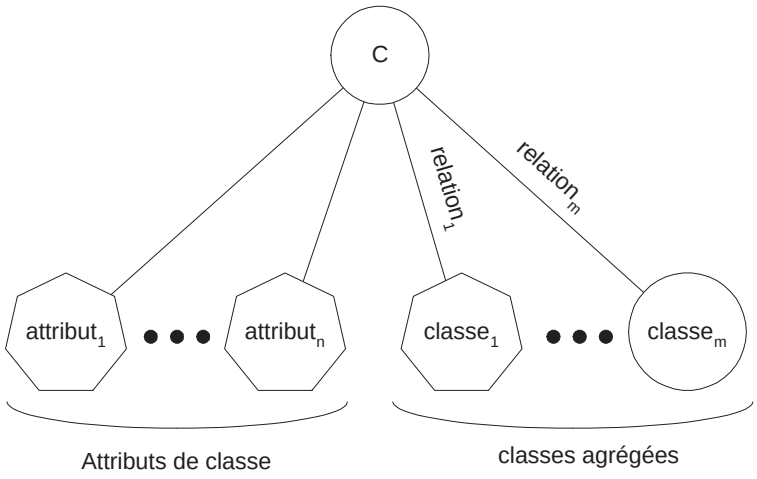

FIG. 3.7 – Arbre xml d'une classe quelconque

Une fois ces deux problèmes résolus, il est possible de définir récursivement un opérateur de similarité entre deux alertes représentées par deux instances de la classe *Alert* définie par l'IDMEF. Pour répondre à ces deux problèmes, nous prenons l'exemple d'une classe C représentée sur la figure 3.7. Cette classe possède n attributs qui sont des types de base et m classes agrégées qui peuvent être des types de base ou des classes. Chaque classe agrégée $classe_i$ possède une relation d'agrégation $relation_i$ avec C. Les classes définies dans le modèle de données IDMEF sont chacun des cas particuliers de cet exemple. Nous allons considérer successivement les différents cas de comparaison d'instances d'attributs de deux instances C_1 et C_2 de C :

- La comparaison des $n' \leq n$ instances d'attributs, instanciés à la fois dans C_1 et C_2, fait directement appel aux fonctions de similarité ($n' \leq n$ car il se peut que certains de ces attributs ne soient pas instanciés).

- La comparaison des instances des classes agrégées $classe_i$ dépend de leur relation $relation_i$ avec C :

  - Si $relation_i = 0...1$: cela veut dire que la classe $classe_i$ est un attribut optionnel. La comparaison est donc possible si C_1 et C_2 possèdent toutes les deux une instance de $classe_i$. Dans ce cas, la fonction de similarité associée est utilisée si $classe_i$ est un type de base. Si ce n'est pas un type de base mais une classe, la valeur de similarité est calculée à partir de l'agrégation des valeurs de similarité des instances d'attributs des deux instances de $classe_i$. Si C_1 et/ou C_2 ne possèdent pas d'instance de $classe_i$, aucune valeur de similarité n'est ni calculée ni prise en compte dans le calcul de la similarité entre C_1 et C_2.

– Si $relation_i = 0...n$: dans ce cas nous pouvons avoir à comparer deux ensembles d'instances de $classe_i$ de cardinalité différente. Selon le type d'entité représenté par $classe_i$ nous en déduisons la nature d'un ensemble d'instances de $classe_i$, qui peut être ordonné ou non-ordonné. Selon la nature ordonnée ou non-ordonnée de cet ensemble nous traitons la comparaison des ensembles de deux manières différentes :

  – Cas non-ordonné : il n'existe pas de relation d'ordre sur les ensembles, aucun élément ne peut être préféré à un autre. Dans ce cas, chaque élément du premier ensemble d'instances doit être comparé avec chaque élément du deuxième ensemble d'instances. Par exemple, dans le cas de la classe $Address$, qui est une classe agrégée de la classe $Node$, un ensemble d'instances de cette classe est non-ordonné. En effet cette classe représente les adresses de réseaux, matériels et applications.

  – Cas ordonné : il existe une relation d'ordre sur les ensembles, il est possible de définir un premier élément et un dernier élément dans chaque ensemble d'instances. Si les deux ensembles ont la même cardinalité, leur comparaison consiste à comparer les éléments ayant les mêmes index dans les liste ordonnées d'instances. Si les deux ensembles n'ont pas la même cardinalité, alors nous utilisons la méthode de calcul du cas non-ordonné. Par exemple, dans le cas de la classe arg, qui est une classe agrégée de la classe $WebService$, un ensemble d'instances de cette classe est ordonné. Un ensemble d'instances de arg représente les arguments passés à un script cgi.

Maintenant que nous avons défini comment comparer deux instances d'une même classe, nous pouvons définir précisément l'opérateur de similarité Sim.

3.2.3 Opérateur de similarité

Dans cette partie nous définissons l'opérateur de similarité Sim. Bien que nous l'appliquons à deux instances de la classe Alert, il est possible d'utiliser Sim pour comparer deux instances d'une même classe faisant partie du modèle IDMEF. Afin de définir précisément Sim, nous reprenons l'exemple de la classe C définie dans la partie précédente. Soient C_1 et C_2 deux instances de C. Avant de calculer l'ensemble des valeurs de similarité, nous déterminons l'ensemble des poids P à utiliser en fonction des événements e_1 et e_2 de $\mathcal{E}_{vt}$ associés aux alertes contenant C_1 et C_2. Si $e_1 = e_2$, nous utilisons l'ensemble des poids $P = p(e)$ associés à $e = e_1 = e_2$. Si e_1 et e_2 sont différents ou correspondent à l'événement $unknown$, P est l'ensemble des poids par défaut.

Nous n'exprimons pas directement $Sim(C_1, C_2)$ mais nous traitons séparément les comparaisons des sous-ensembles d'instances d'attributs de C_1 et C_2 :

– Comparaison des n' attributs (instances de type de base) : la première partie du calcul consiste à sommer les valeurs de similarité entre les n' attributs, pondérées par leur poids. Nous désignons par p_i le poids associé à l'attribut $attribut_i$ et par f_{s_i} sa fonction de similarité. $attribut_{1_i}$ désigne la i^{eme} instance d'attribut de C_1, $attribut_{2_i}$ désigne la i^{eme} instance d'attribut de C_2 :

$$Sim_{types-base}(C_1, C_2) = \sum_{i=0}^{n'} p_i f_{s_i}(attribut_{1_i}, attribut_{2_i})$$

– Comparaison des classes agrégées : pour la comparaison des instances des $classe_i$, nous ne considérons que les instances de $classe_i$ existant à la fois dans C_1 et C_2 pour un i donné. C_1 et C_2 instancient soit l'ensemble des $classe_i$, $i = 0...n$, soit un sous-ensemble. Dans le cas où ils n'instancient qu'un sous-ensemble de ces classes, nous ne comparons que le sous- ensemble des $classe_i$ instanciées en commun. Soient C_{a_1} et C_{a_2} les ensembles d'instances des $classe_i$ de C_1 et C_2, C_{a_1} et C_{a_2} étant les ensembles d'instances du sous-ensemble des $classe_i$ commun à C_1 et C_2. C_{a_1} et C_{a_2} sont composés de sous-ensembles d'instances de la même classe. Ces sous-ensembles sont ordonnés ou non-ordonnés comme nous l'avons vu dans la partie précédente. Soient C_{o_1} et C_{o_2} deux sous-ensembles ordonnés de C_{a_1} et C_{a_2} d'instances de la même classe $classe_n$ et Soient C_{no_1} et C_{no_2} deux sous-ensembles non-ordonnés de C_{a_1} et C_{a_2} d'instances de la même classe $classe_m$. Soient p_m le poids associé à $classe_m$ et f_{s_m} sa fonction de similarité. Nous avons alors :

– $Sim_{non-ordonne}(C_{no_1}, C_{no_2}) =$

$$\frac{1}{|C_{no_1}| + |C_{no_1}|}\{\sum_{i=0}^{|C_{no_1}|} max_{j\in\{1,|C_{no_2}|\}}(f_{s_m}(C_{no_1}[i], C_{no_2}[j])) + \sum_{i=0}^{|C_{no_2}|} max_{j\in\{1,|C_{no_1}|\}}(f_{s_m}(C_{no_2}[i], C_{no_1}[j]))\}$$

$|C_{no_1}|$ et $|C_{no_2}|$ sont les nombres d'éléments de C_{no_1} et C_{no_2} et $C_{no_1}[i]$ est le i^{eme} élément de C_{no_1}. Etant donné que nous comparons deux ensembles non ordonnés, il est nécessaire de comparer chaque élément du premier ensemble avec chaque élément du deuxième. Cependant nous ne considérons pas toutes les valeurs calculées car nous voulons garder certaines propriétés. Par exemple, considérons un ensemble de n instances $E_C = \{C_1, ..., C_n\}$ d'une classe C. Si nous comparons E_C avec lui-même, il est souhaitable que nous ayons $Sim_{non-ordonne}(E_C, E_C) = 1$. Ainsi si nous avons $f_s(C_i, C_i) = 1$ et $f_s(C_i, C_j) < 1$ pour $i \neq j$ (les éléments de l'ensemble ne sont pas similaires entre eux), nous avons bien $Sim_{non-ordonne}(E_C, E_C) = 1$. En revanche, si nous faisons la somme des valeurs calculées et que nous la divisons par le nombre de valeurs calculées, nous obtenons dans l'exemple considéré $s = \frac{1\times n+n(n-1)\times\epsilon}{n^2}$, ϵ étant la moyenne des $f_s(C_i, C_j)$ pour $i \neq j$. Dans ce cas là le résultat s de la comparaison de E_C avec lui-même est inférieur à 1.
Pour éviter ce problème, pour chaque élément de C_{no_1} nous ne considérons

que la valeur maximale calculée lors de sa comparaison avec les éléments de C_{no_2}. Pareillement, lors de la comparaison de chaque élement de C_{no_2} avec ceux de C_{no_1}, nous ne retenons que les valeurs maximales calculées.

– $Sim_{ordonne}(C_{o_1}, C_{o_2}) = \frac{1}{|C_{o_1}|} \sum_{i=0}^{|C_{no_1}|} f_{s_m}(C_{no_1}[i], C_{no_2}[i])$
si $|C_{o_1}| = |C_{o_2}|$. Sinon nous appliquons le cas non-ordonné.

Application de la pondération

En regroupant les trois termes présentés ci-dessus et en pondérant les valeurs de similarité, on obtient l'expression suivante pour l'opérateur de similarité Sim :

$$Sim(C_1, C_2) = \frac{1}{\sum_k p_k} \{ \overbrace{\sum_{i=0}^{n} p_i f_{s_i}(attribut_{1_i}, attribut_{2_i})}^{S_1} + \overbrace{\sum_{a \in NO(C_1), b \in NO(C_2)} p_{NO_{ab}} \times Sim_{non-ordonne}(a, b)}^{S_2} + \overbrace{\sum_{a \in O(C_1), b \in O(C_2)} p_{O_{ab}} \times Sim_{ordonne}(a, b)}^{S_3} \}$$

où le le terme S_1 représente les comparaisons des attributs de classe instanciés dans C_1 et C_2 pondérés par leur poids p_i. Le terme S_2 représente les comparaisons des sous-ensembles d'instances d'attributs de C_1 et C_2 non-ordonnés, $NO(C)$ désignant l'ensemble des sous-ensembles non-ordonnés d'instances d'attributs d'une instance d'une classe C et $p_{NO_{ab}}$ étant le poids associé au type des classes comparées. Dans S_1, a et b désignent des sous-ensembles d'instances, tout comme C_{no_1} et C_{no_2} que nous avons utilisé plus haut. De la même manière, S_3 représente les comparaisons des sous-ensembles d'instances d'attributs de C_1 et C_2 ordonnés, $O(C)$ désignant l'ensemble des sous-ensembles ordonnés d'instances d'attributs d'une instance d'une classe C et $p_{O_{ab}}$ étant le poids associé au type des classes comparées. $\sum_k p_k$ désigne l'ensemble des poids ayant été utilisé dans le calcul des différents termes. La division par ce terme permet d'obtenir une valeur de similarité dans $[0, 1]$.

Nous pouvons maintenant comparer deux instances d'une même classe faisant partie du modèle de données IDMEF, il nous reste cependant à définir les fonctions de similarité .

3.2.4 Fonctions de similarité

Cette partie présente les fonctions de similarité que nous avons définies pour implémenter l'opérateur de similarité. Nous n'avons pas défini de fonction de si-

milarité sur l'ensemble des attributs définis dans le modèle de données IDMEF, certains de ces attributs n'étant pas pertinents pour une mesure de similarité. Par exemple, dans la classe $OverflowAlert$, un attribut optionnel $buffer$ représente le buffer utilisé pour le dépassement de capacité. Selon le SDI utilisé, la quantité de données capturée peut varier. En conséquence, étant donné que la mesure de similarité entre deux buffers consisterait à vérifier s'ils sont égaux, il est préférable de ne pas prendre en compte cette information. En effet, nous pourrions rencontrer le cas où l'on compare deux buffers correspondants à la même attaque, mais capturés de manière différente par deux SDI différents. Une autre possibilité serait de chercher le ratio de données communes entre les deux buffers, mais il serait alors possible de considérer que deux buffers sont similaires alors que l'un correspond à une variante de l'attaque utilisant une partie du code. Dans ce cas, on risque d'agréger des attaques similaires au niveau de leur principe, mais pas forcément au niveau de leur exécution.

Un autre exemple est celui de la classe $Classification$, qui possède la classe agrégée $Reference$. La classe $Reference$ contient des informations permettant d'obtenir plus de détail sur l'événement associé à l'alerte. Une instance de cette classe contiendra par exemple un lien vers le site CVE et une description textuelle de l'événement. Comparer de telles informations n'est pas pertinent et ne nous donne pas d'information sur la similarité entre deux alertes.

La classe IDMEF qu'instancient deux alertes est la classe *Alert*. Nous ne traitons pas le cas où un message IDMEF est une instance de la classe $HeartBeat$, cette classe servant à rapporter l'état d'un SDI. La classe *Alert* possède neuf classes agrégées (voir 2.5.2). Nous allons exposer comment nous comparons les différents attributs de ces classes. Avant d'aborder la liste des classes à comparer, il est judicieux de considérer le cas des chaînes de caractères. Certains attributs sont des chaînes de caractères, sans formatage particulier. En fonction de la sémantique associée à ce type d'attribut nous décidons de faire une simple comparaison d'égalité entre deux chaînes de caractères donnant une similarité de 1 ou 0, ou alors nous ignorons l'attribut.

- Classe *Analyzer* :
  Une alerte ne contient qu'une instance de cette classe. Elle permet d'identifier le SDI à l'origine de l'alerte. Ces informations ne sont pas en rapport avec l'événement associé à l'alerte, en conséquence nous ne les utilisons pas.

- Classes $Createtime$ et $DetectTime$:
  Ces classes nous donnent les informations temporelles liées à l'événement détecté. La date qui nous intéresse le plus dans la comparaison d'alertes est l'instant de la détection de l'événement (instance de $DetectTime$). En effet, la date de création de l'alerte est obligatoirement présente mais ne correspond pas forcément à la date de détection. La date de création est forcément plus ancienne ou égale à la date de détection.

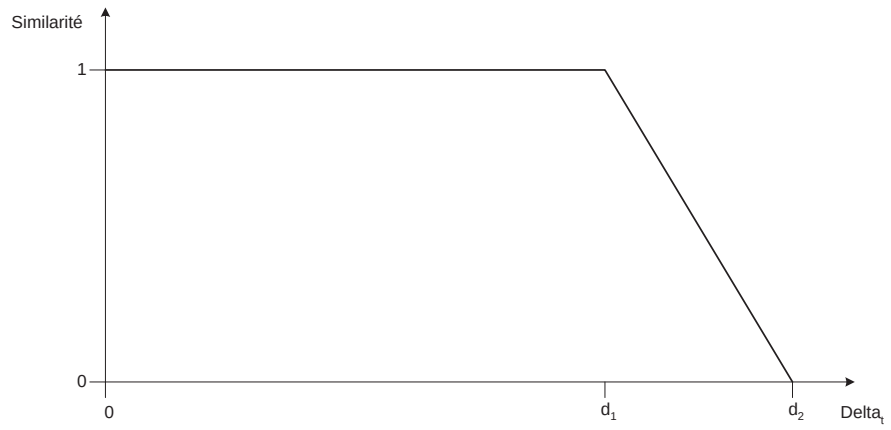

FIG. 3.8 – Fonction de similarité temporelle. L'axe des abscisses représente la valeur absolue de la différence de temps entre deux alertes à comparer.

Afin de comparer deux dates, nous proposons de nous baser sur des mesures expérimentales pour chaque événement de l'ensemble $\mathcal{E}_{vt}$. Pour chaque événement de $\mathcal{E}_{vt}$, nous exécutons l'événement sur un réseau où sont placés plusieurs SDI. L'exécution de l'événement se fait dans deux environnements différents : une première fois sans trafic additionnel sur le réseau et une deuxième fois avec du trafic simulant un réseau chargé. Nous mesurons alors le temps écoulé entre la réception de la première et de la dernière alerte générée. Ainsi, nous obtenons deux intervalles de temps d_1 et d_2. Ces deux intervalles nous permettent de définir une fonction de similarité temporelle pour chaque événement de $\mathcal{E}_{vt}$. Grâce à cette fonction nous pouvons calculer la similarité entre deux instances de classe $Createtime$ ou $DetectTime$ qui sont alors en fait des instances du type *ntpstamp*. L'idée est de soustraire les deux dates des alertes à comparer pour obtenir un délai que nous comparons aux délais d_1 et d_2. Si le délai calculé est inférieur à d_1, les dates des événements sont similaires et s'il est entre d_1 et d_2 la similarité décroît à mesure que l'on s'approche du délai maximum d_2. Au delà de d_2 nous considérons que les alertes ne sont pas relatives au même événement. Cette fonction est représentée sur la figure 3.8.

– Classes $Source$ et $Target$:
Ces classes ont une structure similaire et plusieurs de leurs attributs peuvent être traités par les même fonctions de similarité. Nous allons passer en revue les différentes classes agrégées et leurs attributs. Les classes $Source$ et $Target$ ont trois attributs. Deux de ces attributs ne sont pas pertinents pour la comparaison, l'un étant un identifiant unique dont la valeur dépend de la quantité d'information présente dans l'alerte (voir la définition des « unique identifiers » dans [27]) et l'autre identifiant une interface du SDI ayant émis l'alerte.
Il nous reste à étudier les classes impliquées dans la comparaison d'instances des classes $Source$ et $Target$. Nous regroupons l'étude de ces deux classes car elles possèdent une structure similaire, la classe $Target$ possédant une classe

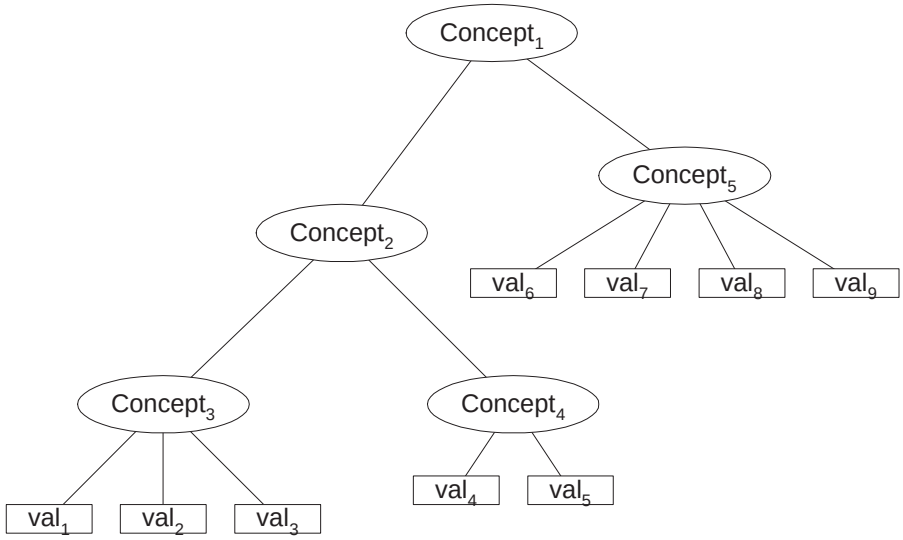

FIG. 3.9 – Exemple d'arbre de taxonomie arbitraire

agrégée supplémentaire : la classe $FileList$.

– Classe $Node$: les deux attributs *ident* et *category* ne sont pas utilisés pour la comparaison car ce sont des données non reliées directement à l'événement. Les classes *location* et *name* sont des chaînes de caractères et permettent d'identifier le matériel impliqué dans l'événement. Elles sont donc prises en compte dans le calcul de similarité. Nous détaillons ci-dessous la comparaison des instances de la classe *Address* qui est une classe agrégée de la classe $Node$.

– Classe *Address* : lors de la comparaison de deux instances de cette classe nous nous interessons aux informations concernant deux adresses. L'attribut *category* nous permet de savoir comment est formatée l'adresse mais n'est pas comparé avec une autre instance d'une autre alerte. L'information la plus importante est contenue dans une instance de l'attribut *address*. Afin de mesurer la similarité entre deux adresses, nous utilisons une taxonomie du réseau surveillé, comme proposé dans le travail de Klaus Julisch ([59]). Cependant Julisch quantifie dans quelle mesure deux alertes sont différentes. Nous devons ici définir comment nous utilisons une taxonomie pour calculer une mesure de similarité. Pour ce faire, nous prenons l'exemple d'une taxonomie quelconque représentée sur la figure 3.9.
Afin de mesurer la similarité entre deux feuilles de cet arbre, nous proposons de calculer le ratio de branches communes aux deux valeurs par rapport au chemin le plus long pour atteindre l'une des deux valeurs. Par exemple, pour atteindre la valeur val_1 il faut traverser 3 branches, pour atteindre la valeur val_2 nous parcourons 3 branches dont 2 sont en commun avec le chemin vers val_1. Dans ce cas on a donc une valeur de similarité de $\frac{2}{3}$. Entre val_1 et val_4

la similarité vaut $\frac{1}{3}$ et entre val_1 et val_6 la similarité vaut 0.
La classe *Address* possède une relation d'agrégation de type 0...∗ avec la classe *Node*. Nous considérons qu'un ensemble d'instance de la classe *Address* est non-ordonné.

- Classe $User$: cette classe contient des données sur des utilisateurs liés à l'événement associé à l'alerte. Comparer deux instances de cette classe consiste à comparer deux ensembles d'instances de la classe $UserId$. Comparer deux instances de la classe $UserId$ consiste à comparer les noms et les numéros d'utilisateur associés aux personnes désignées par ces instances. Ces deux données étant respectivement une chaîne de caractères et un numéro, nous testons tout simplement leur égalité entre deux instances afin de produire les valeurs de similarité. Lors de la comparaison de deux ensembles d'instances de la classe $UserId$, nous les considérons non ordonnés.

- Classe *Process* : cette classe encapsule les informations liées à un processus impliqué dans l'événement associé à l'alerte. La comparaison d'instances des classes *name*, *pid* et *path* consiste en un test d'égalité. En revanche pour les classes *arg* et *env*, étant donnée leur relation d'agrégation avec la classe *Process*, il se peut que deux ensembles d'instances de ces classes soient comparés. Dans ce cas, la comparaison de chaque instance se fait via un test d'égalité et les ensembles sont considérés ordonnés.

- Classe *Service* : dans le cas de la comparaison de deux instances de cette classe, l'information pertinente à comparer est l'information sur le ou les numéros de port associés à l'événement. Ainsi, nous avons à comparer éventuellement deux instances de *name* donnant le nom du port. Dans ce cas, un test d'égalité entre chaînes de caractères est effectué. Il en va de même si *port* est instancié, un seul numéro de port étant spécifié dans les deux instances comparées. En revanche dans le cas de la comparaison de deux instances de *portlist*, nous comparons deux ensembles non ordonnés.
  Dans le modèle de données IDMEF, la classe *Service* est décomposée en deux classes filles $WebService$ et $SNMPService$. Cela se traduit par la comparaison d'attributs supplémentaires dans le cas de la comparaison de deux instances de ces classes filles :

  - Classe $WebService$: les classes supplémentaires concernent le trafic web associé à l'événement. Les trois premières classes, *url*, *cgi* et *http-method*, sont des chaînes de caractères. Nous prenons en compte les comparaisons de leurs instances quand ces attributs sont instanciés. La quatrième classe agrégée, arg, a une relation d'agrégation 0...∗ avec la classe $WebService$. Un ensemble d'instances de la classe arg est considéré comme ordonné.

  - Classe $SNMPService$: les attributs supplémentaires sont tous du type

chaîne de caractères. La comparaison des instances de ces classes se fait par un test d'égalité.

- Classe *FileList* : cette classe n'est éventuellement instanciée que dans la classe *Target*. La classe *File* possède une relation d'agrégation 1...∗ avec la classe *FileList*. Un ensemble d'instances de la classe *File* est considéré ordonné. La classe *File* contient les différents attributs d'un fichier. Pour chacun des attributs obligatoires ou ayant une relation 0...∗ avec la classe *File*, nous testons l'égalité de deux instances de ces attributs pour le calcul de la similarité. Les instances de la classe *Checksum* ne sont pas utilisées dans le calcul de similarité. Les ensembles d'instances de classe *FileAccess* et *Linkage* sont considérés ordonnés.

- Classe *Classification* : l'information importante est contenue dans une instance de l'attribut *text*. Cet attribut est une chaîne de caractères identifiant le nom de l'événement attaché à l'alerte. C'est grâce à cet attribut et au dictionnaire d'événement que nous pouvons sélectionner les poids à utiliser lors du calcul de similarité.

Les fonctions de similarité étant spécifiées, il nous reste à définir comment nous utilisons les valeurs de similarité calculées entre les alertes pour les regrouper en ensembles d'alertes semblables.

3.2.5 Algorithmes d'agrégation

Le problème du partitionnement d'un ensemble d'éléments en sous-ensembles d'éléments similaires est un problème bien connu ayant déjà été traité dans plusieurs domaines de recherche. Notre but étant de traiter les alertes en temps réel, ou du moins dans une approche en-ligne, notre choix des algorithmes développés s'en trouve restreint.

Dans [59], Julisch utilise un algorithme s'appliquant à un ensemble d'alertes traitées hors-ligne. Nous rappelons que dans son travail, Julisch recherche des groupes d'alertes émises par la détection d'événements liés à des problèmes de configuration du système informatique surveillé. Cet algorithme consiste à trouver parmi un ensemble d'alertes, un sous-ensemble dont la taille est supérieure à un entier donné, et dont l'hétérogénéité (définie à partir de la notion de dissimilarité) est minimale. Une fois un sous-ensemble d'alertes similaire trouvé, l'algorithme doit être réitéré sur l'ensemble d'alertes restantes et ainsi de suite. Cet algorithme est non supervisé.

Dans [86], les auteurs définissent une mesure de similarité entre deux alertes et proposent un algorithme de regroupement en temps réel. Cet algorithme consiste à ajouter une alerte candidate au groupe d'alertes le plus similaire si cette valeur de similarité maximale est au-dessus d'un seuil fixé par l'administrateur.

Avant d'exposer les algorithmes que nous avons adoptés et implémentés, nous allons rappeler les principales approches de regroupement. Le regroupement consiste

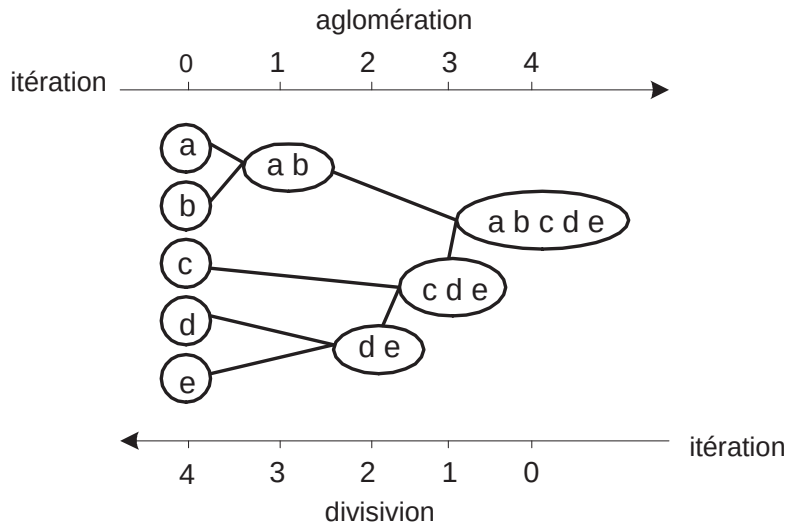

FIG. 3.10 – Principe des algorithmes hiérarchiques

à construire des groupes d'objets à partir d'un ensemble d'objets, ces objets étant similaires au sein d'un même groupe et dissimilaires s'ils appartiennent à des groupes différents. Le regroupement est un processus non supervisé, c'est à dire que l'on ne connaît pas à priori le nombre de groupes à former. Les méthodes de regroupement, ou de clustering, se divisent en plusieurs catégories ([50, 56, 16]) :

- Algorithmes de partitionnement ([56]) : ils consistent à construire plusieurs partitions puis à les évaluer selon certains critères. Plus précisément, le principe est de construire une partition de k "clusters" d'une base D de n objets, les k "clusters" devant optimiser le critère choisi.

- Algorithmes hiérarchiques ([58]) : ils consistent à créer une décomposition hiérarchique des objets selon certains critères. La matrice des distances entre les objets est utilisée comme critère de regroupement et une condition d'arrêt doit être spécifiée, par exemple une distance maximale ou minimale selon la méthode utilisée. En effet selon que la méthode aglomére ou divise (voir figure 3.10), la condition d'arrêt est un maximum ou un minimum.

- Algorithmes basés sur la densité ([16]) : ce sont des algorithmes basés sur des notions de connectivité et de densité. Les "clusters" sont vus comme des régions denses séparées par des régions qui le sont moins (bruit).

- Algorithmes de grille ([51]) : ce sont des algorithmes basés sur une structure à multi-niveaux de granularité.

Nous avons choisi d'implémenter deux algorithmes : un algorithme destiné au traitement en ligne et un second basé sur une méthode hiérarchique. Nous n'avons

pas choisi les algorithmes de type k-mean ([66], algorithme de partitionnement) car ils nécessitent de spécifier le nombre de "clusters" k.

Le premier algorithme implémenté est similaire à celui utilisé par Alfonso Valdes et Keith Skinner dans [86]. Le principe est le suivant : à chaque fois qu'une nouvelle alerte est reçue par le module d'agrégation, elle est comparée aux "clusters" déjà formés. Nous considérons alors le "cluster" le plus similaire à la nouvelle alerte, si la valeur de similarité est supérieure au seuil de similarité fixé, nous ajoutons l'alerte au "cluster". Si la valeur de similarité est inférieure au seuil, nous créons un nouveau "cluster" contenant la nouvelle alerte. Cet algorithme nécessite donc de spécifier un seuil. Nous ne connaissons pas de méthode ou d'heuristique nous permettant de trouver une "bonne" valeur de ce seuil. C'est donc par l'expérimentation (voir chapitre 6, section 6.1) que nous avons déterminé une valeur de ce seuil donnant de bons résultats. Un seuil de 0.8 nous a permis d'obtenir des résultats cohérents.

Le deuxième algorithme implémenté est un algorithme hiérarchique. L'outil d'agrégation que nous avons réalisé peut fonctionner selon deux modes : en ligne en utilisant l'algorithme précédent, et hors-ligne en traitant un ensemble d'alertes. En mode hors-ligne l'algorithme est le suivant : nous calculons tout d'abord la matrice de similarité entre les alertes puis nous regroupons les deux éléments les plus similaires en cherchant la valeur de similarité maximale dans la matrice. Ensuite la matrice de similarité est partiellement réévaluée pour ensuite réitérer la phase de regroupement. La condition d'arrêt de cet algorithme est une condition sur la valeur de similarité maximale de la matrice. Si la valeur de similarité maximale de la matrice est inférieure à une valeur de similarité seuil spécifiée au début du traitement, l'algorithme est stoppé.

Dans les parties précédentes, nous avons exposé comment regrouper des alertes selon un critère de similarité dont nous avons aussi exposé l'évaluation. Maintenant que nous savons former des groupes d'alertes similaires, nous allons voir comment fusionner les informations d'un groupe d'alertes afin d'obtenir une alerte synthétique équivalente, en terme d'informations, au groupe d'alertes initial.

3.3 Fusion d'alertes

L'opération de fusion des informations d'un groupe d'alertes similaires est la dernière opération effectuée par le module d'agrégation/fusion avant d'envoyer l'alerte, dite 'de fusion', aux autres modules de traitement. Outre l'opération de fusion des informations, nous devons répondre à d'autres problèmes liés au transfert des alertes de fusion aux modules suivants dans la chaîne de traitement.

Nous commençons par présenter la méthode développée pour la fusion d'alertes similaires puis nous nous penchons sur les problèmes de transfert des alertes de fusion vers d'autres modules de traitement. Nous n'abordons pas ici la question du stockage de ces alertes dans une base de données.

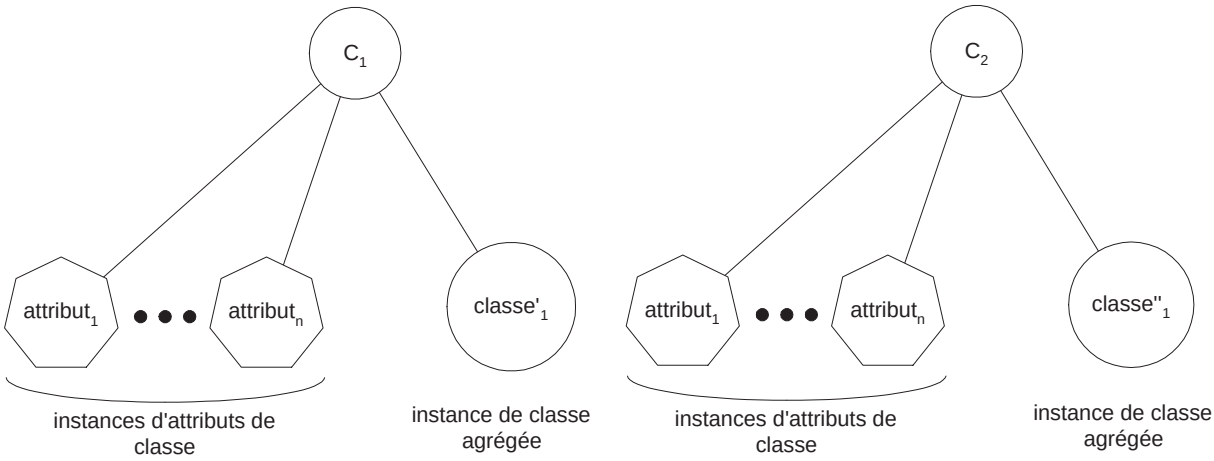

FIG. 3.11 – Exemple de deux instances C_1 et C_2 de C

3.3.1 Fusion d'informations entre deux alertes

Afin d'illustrer l'opération de fusion des informations d'un groupe d'alertes, nous prenons l'exemple d'une classe quelconque représentant le cas général d'une classe du modèle de données IDMEF. Étant donné que nos alertes sont modélisées par des instances de la classe *Alert* du modèle de données IDMEF, le problème de fusion d'alertes se ramène au problème de fusion d'instances de classes IDMEF. Nous reprenons l'exemple que nous avons présenté en 3.2.2. Prenons le cas de deux instances C_1 et C_2 de cette classe. C_1 et C_2 sont égales si les ensembles des instances de leurs attributs sont égaux. Mais étant donné que certaines classes agrégées peuvent avoir une relation du type $0...*$, il est possible que C_1 et C_2 soient partiellement égales. Pour illustrer ce cas de figure, prenons l'exemple de la figure 3.11 où sont représentées deux instances de C.

Nous considérons que les n attributs de classe sont instanciés dans chaque instance de C et que nous avons $C_1.attribut_i = C_2.attribut_i$ pour $i \in [0...n]$. D'autre part nous considérons qu'une instance de classe agrégée $classe_1$ existe dans C_1 et C_2 et que $C_1.classe'_1 \neq C_2.classe''_1$. Nous faisons aussi l'hypothèse que la relation de $classe_1$ avec C est $0...*$. Dans ce cas, nous avons $C_1 \neq C_2$ mais, étant donné que la relation de $classe_1$ avec C est $0...*$, nous pouvons créer une instance C_3 de C factorisant C_1 et C_2 (figure 3.12).

Cet exemple illustre la façon dont nous procédons pour fusionner des instances de la classe *Alert* du modèle de données IDMEF. Nous allons maintenant exposer comment cette opération est appliquée à deux instances de la classe *Alert*.

Les deux classes agrégées ayant une relation $0...*$ avec la classe *Alert* sont les classes $Source$ et $Target$. Si nous comparons deux instances d'une de ces classes, nous pouvons donc ajouter autant d'instances que nous désirons s'il n'est pas possible de les fusionner. Certaines classes du modèle de données IDMEF sont des conteneurs. Les classes $FileList$ et $User$ sont de ce type. Une alerte contient au plus une instance de la classe $FileList$ qui contient autant d'instances de la classe $File$ qu'il y

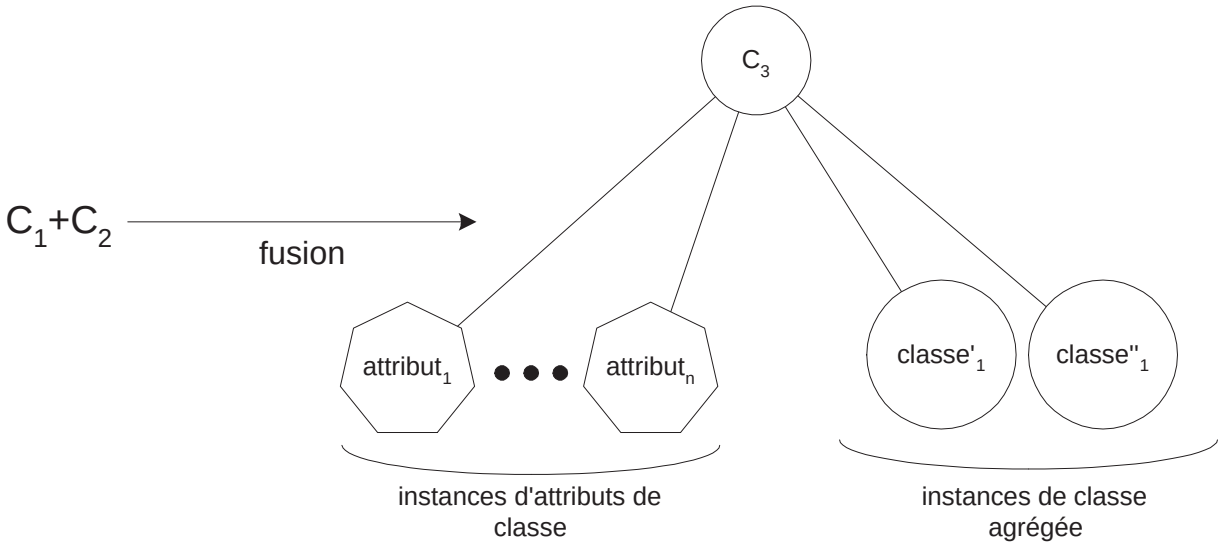

FIG. 3.12 – Exemple de factorisation de C_1 et C_2 en C_3

a de fichiers concernés par l'événement associé à l'alerte. Il en va de même pour la classe $User$ qui contient autant d'instances de la classe $UserId$ que nécessaire. La fusion de deux instances de la classe $User$ consiste à fusionner les listes d'instances de $UserId$ si le format des données est le même. Si le format n'est pas le même, nous pouvons simplement concaténer les deux listes d'instances de $UserId$. L'information permettant de déterminer le format des données des instances de $UserId$ est contenue dans l'attribut *category*.

3.3.2 Transfert des alertes de fusion au module de corrélation

Nous exposons ici les choix d'implantation faits dans le cadre du dialogue entre le module d'agrégation/fusion et le module de corrélation. Étant donné qu'un groupe d'alertes est constitué de plusieurs alertes n'ayant pas été reçues en même temps, nous devons décider au bout de combien de temps après la réception de la première alerte nous fusionnons les informations. Notre implantation effectue le processus de fusion dès que le délai d_2 défini précédemment (section 3.2.4) et dépendant de chaque événement de $\mathcal{E}_{vt}$ est expiré après avoir reçu la première alerte d'un groupe. Un tel groupe d'alerte est alors qualifié de *stable*. L'alerte de fusion résultante est alors envoyée au module de corrélation. Cependant, pour certains événements le délai d_2 peut être long, de l'ordre de plusieurs secondes ou plus. Dans ce cas, attendre que le groupe d'alertes devienne stable induit un retard dans le processus de détection. Nous avons donc choisi dans notre implantation de proposer deux modes de fusion et de dialogue avec le module de corrélation. Le premier mode a été décrit au début de ce paragraphe et consiste à envoyer une alerte de fusion dès que d_2 a expiré. Le deuxième mode, associé aux événements dont la valeur d_2 est grande, consiste à fusionner les alertes au fur et à mesure qu'elles sont regroupées et à envoyer cette alerte de fusion intermédiaire au module de corrélation. Ainsi, le module de corrélation

reçoit des alertes dites *de mise à jour* permettant de mettre à jour les alertes de fusion. Afin de distinguer ces alertes des alertes de fusion normales, nous utilisons le champ *additionaldata* en ajoutant un attribut appelé *AlertType*. Plus de détails sont donnés dans le chapitre 6 où nous exposons l'implantation de notre approche de l'agrégation et de la fusion d'alertes ainsi que les résultats expérimentaux.

3.3.3 Conclusion

Nous avons présenté le problème des alertes redondantes dans le contexte de la détection d'intrusions coopérative. Ce problème est important dans la mesure ou l'absence de traitement du flux d'alertes généré par un ensemble de SDI ne permet pas de fournir un diagnostic exploitable par l'administrateur système. D'autre part, une trop grande quantité d'alertes peut submerger un module de traitement d'alertes et rendre ce dernier indisponible. Ainsi, plusieurs approches ont été proposées afin de réduire le volume de données à analyser par l'administrateur sans supprimer d'informations. Les différentes approches présentées se proposent de réunir les alertes générées lors de la détection du même événement. Pour ce faire, dans [29] les auteurs définissent des règles de regroupement. Dans [59] et [86], les auteurs définissent respectivement une notion de dissimilarité et de similarité afin de regrouper des alertes en des ensembles d'alertes similaires. Dans [18] ce sont des prédicats logiques qui permettent de déterminer si deux alertes sont similaires. Nous avons choisi de définir un opérateur de similarité utilisant des fonctions de similarité entre attributs d'alerte afin de pouvoir rassembler les alertes en groupes d'alertes similaires. Nous avons présenté comment nous fusionons les informations de ces groupes d'alertes afin de produire des alertes plus synthétiques. Outre le fait de présenter à l'administrateur une information plus concise sous la forme d'alertes de fusion, le mécanisme d'agrégation et de fusion d'alertes constitue un pré-traitement des alertes indispensable avant tout autre traitement demandant une certaine puissance de calcul. L'opérateur de similarité et la fonction de fusion ont été définies sur le format IDMEF pour faciliter l'insertion d'un module implantant ces fonctionalités dans une architecture coopérative. L'implantation que nous avons réalisée est présentée dans le chapitre 6 ainsi que les résultats expérimentaux obtenus.

Chapitre 4

Corrélation d'alertes

La notion de corrélation d'alertes a été abordée dans plusieurs articles mais sa sémantique n'est pas unique. Avant de présenter les travaux existants traitant de la notion de corrélation et avant de définir la corrélation telle que nous l'utilisons, nous rappelons les problèmes que nous voulons résoudre grâce à cette technique.

Les fonctionnalités d'agrégation et de fusion décrites dans le chapitre précédent nous permettent de réduire le nombre d'alertes à traiter en éliminant les informations redondantes et en rassemblant les alertes similaires. Le flux d'alertes résultant de ce traitement doit être encore traité afin d'extraire les informations concernant les scénarios d'attaques en cours de réalisation. En effet, nous faisons l'hypothèse qu'un attaquant développe une stratégie d'attaque et que la mise en œuvre de cette stratégie se traduit au niveau des SDI existants par la génération d'un ensemble d'alertes de détection d'intrusions.

4.1 La notion de corrélation d'alertes

Dans cette partie nous présentons les différents travaux relatifs à la notion de corrélation d'alertes de détection d'intrusions. La corrélation d'alertes telle que nous allons la définir vise à détecter les scénarios d'attaque en cours de réalisation sur un système informatique. Le type de corrélation que nous définissons est dite semi-explicite et repose sur la découverte de liens de corrélation entre modèles d'attaques. Il existe deux autres types de corrélation, la corrélation implicite et la corrélation explicite. Dans le cas de la corrélation implicite, les relations de corrélation entre événements peuvent être extraites d'un ensemble de données. L'approche explicite de la corrélation consiste à identifier les relations de corrélation entre des modèles d'événements pour construire une base de plans d'événements. Dans ce cas, il faut expliciter l'ensemble des plans ou scénarios d'événements réalisables à partir des événements élémentaires.

Nous présentons les travaux se rapportant à la corrélation d'alertes en respectant cette classification. Dans une première partie nous présentons les travaux relatifs à la corrélation implicite, ensuite nous nous intéressons aux travaux utilisant la corrélation explicite puis nous terminons par les travaux relatifs à la corrélation

semi-explicite.

4.1.1 Corrélation implicite

Les fonctionnalités de l'approche implicite de la corrélation d'événements sont différentes des fonctionnalités des approches explicites et semi-explicites. En effet, la fonction principale de ces deux dernières approches est la reconnaissance de plans alors que la corrélation implicite regroupe des événements ayant un lien implicite. Ce lien implicite est révélé par l'exploitation de données ou par la spécification d'un critère de similarité. Ainsi l'approche présentée dans [86], que nous avons détaillée dans la section 3.1.3, se classe dans la catégorie de la corrélation implicite. Nous avons vu en 3.1.3 que par alertes corrélées les auteurs désignent des alertes relatives à la détection du même événement. Cette définition de la corrélation ne correspond pas à celle que nous donnons dans ce chapitre, à savoir que deux événements A et B sont corrélés si, par exemple, les conséquences de l'occurrence de A favorisent la réalisation de B. Nous rappelons que les auteurs utilisent une mesure de similarité entre alertes afin de regrouper les alertes similaires. Cette mesure de similarité permet de regrouper les alertes relatives au même événement. Cependant, les auteurs proposent de relaxer la contrainte sur la valeur de similarité minimale attendu pour la classe de l'événement associé à l'alerte. De cette manière ils prétendent pouvoir agréger des alertes ne correspondant pas au même événement mais faisant partie d'un même scénario d'attaque composé de plusieurs étapes. On peut objecter que les événements relatifs à une machine particulière ne sont pas forcément liés entre eux même si leurs attributs concernent la même machine. Ceci fait apparaître la limite de ce type d'approche dans le cadre de la reconnaissance de plan. Une approche implicite permettra plutôt de découvrir des liens entre alertes générées par plusieurs SDI dans une certaine configuration. Par exemple il est possible d'étudier les alertes générées sur un réseau donné, avec une certaine configuration de sonde, lorsque des attaques sont rejouées. Ainsi, on peut découvrir quelles alertes nous sommes susceptibles d'observer lors de la détection d'un événement afin de pouvoir les regrouper par la suite.

4.1.2 Corrélation explicite

Dans cette section nous présentons les approches existantes de la corrélation explicite. Nous présentons tout d'abord les approches se basant sur un système de règles puis nous abordons les approches basée sur des langages de scénarios.

Systèmes à base de règles

Dans la section 3.1.1 nous avons vu que dans [29] les auteurs proposent plusieurs algorithmes permettant d'agréger et de corréler en temps réel des alertes de détection d'intrusions. Nous avons précédemment étudié l'algorithme permettant de regrouper des alertes considérées comme des duplicatas à partir de règles spécifiées par un

opérateur. Ces règles sont utilisées par l'algorithme de corrélation décrit par les auteurs. Un autre type de règle est proposé que nous présentons plus loin. Ce deuxième type de règles permet d'exprimer les liens causaux entre événements associés aux alertes. L'opérateur définit dans ce cas une *chaîne de conséquences* qui est une liste d'alertes devant être reçues dans un certain ordre et dans une certaine période de temps. Une définition de conséquence est composée des éléments suivants :

- **Classe de l'alerte initiale** : classe de l'alerte initiant la règle.
- **Sonde initiale** : précise le nom de la sonde qui doit émettre l'alerte initialisant la règle. Il est possible de ne pas préciser ce nom afin de ne pas tenir compte de la sonde.
- **Classe de l'alerte conséquence** : classe de l'alerte qui est la conséquence de l'alerte initiale.
- **Sonde détectant la conséquence** : précise le nom de la sonde qui doit émettre l'alerte conséquence d'une autre attaque. Il est possible de ne pas préciser ce nom afin de ne pas tenir compte de la sonde.
- **Gravité** : niveau de gravité donné à l'alerte générée quand l'alerte conséquence n'est pas observée.
- **Durée d'attente** : temps séparant l'alerte initiant la règle et l'alerte représentant la conséquence d'une autre étape du scénario.

Lorsque une nouvelle alerte est reçue, la liste des définitions de conséquence est parcourue pour voir si l'alerte reçue est de la même classe que la classe de l'alerte conséquence définie dans la liste. Si c'est le cas, une alerte de la classe initiale spécifiée par la définition est recherchée dans l'ensemble des alertes déjà reçues. Si une telle alerte est trouvée, la réalisation de la définition de la conséquence est enregistrée. Ce système ne permet pas de spécifier directement un scénario composé de plus de deux étapes. Cependant la spécification d'un ensemble de définitions de conséquence permet de tenir compte de l'ensemble des étapes composant un scénario.

Dans [49], les auteurs définissent une architecture d'analyse de fichiers de log, appelée ASAX. Le but poursuivi est de créer un système facilement utilisable sur différentes plateformes. L'utilisation de ASAX est facilitée par la définition d'un format de fichier de log, NADF, suffisamment générique pour pouvoir convertir les formats existants dans ce format. Afin de trouver dans les logs applicatifs des séquences d'événements constituant un scénario d'attaque, les auteurs définissent le langage RUSSEL. Ce langage a été développé pour traiter efficacement de gros fichiers séquentiels d'audit tout en conservant la simplicité d'un langage à base de règles. Il peut être vu comme un langage incluant une structure de contrôle s'appuyant sur un système de déclenchement de règles. La figure 4.1 montre un exemple de règle écrite en RUSSEL comptabilisant le nombre de tentative de login infructueux.

```
rule Failed_login(maxtimes, duration:integer)
# Cette règle détecte un premier login non réussi et déclenche une
# règle permettant de compter le nombre de tentatives en prenant en
# compte un temps d'expiration
begin
if evt=login and res=failure and is_unsecure(terminal)
-> Trigger off for next Count_rule1(maxtimes-1, timestp+duration)
fi ;
Trigger off for next Failed_login(maxtimes, duration)
end
```

FIG. 4.1 – Exemple de règle écrite dans le langage RUSSEL

Ce langage permet aussi de trouver des scénarios représentés par une succession de commandes. La règle présentée sur la figure 4.2 vise à détecter un scénario constitué par une séquence d'événements enregistrée dans un tableau (*scenario*).

Le langage RUSSEL est intéressant dans la mesure où il définit une structure de contrôle en conservant la simplicité d'un langage à base de règles. Les auteurs proposent de définir un langage de plus haut niveau, RUSSEL2, permettant d'écrire des règles plus facilement. Le langage n'est pas défini dans l'article mais est présenté comme un travail futur.

Langages de scénarios

Dans [70], Hervé Debar et Benjamin Morin proposent d'utiliser les chroniques [37, 26] pour reconnaître des scénarios d'attaque. Le système de reconnaissance de chroniques vise à donner une interprétation de l'évolution du monde étant donné des événements datés. Ce système prend en entrée un flux d'événements datés et reconnaît des instances de chroniques au fur et à mesure qu'elles se développent. Chaque chronique peut être vue comme un ensemble de motifs événementiels sur lesquels un ensemble de contraintes contextuelles et temporelles s'appliquent. Si les événements observés correspondent aux motifs de la chronique et si leur occurrence ne viole pas les contraintes temporelles et contextuelles, alors une instance de la chronique est reconnue. Dans le cas du système de corrélation d'alertes présenté par les auteurs, les événements d'entrée sont les alertes de détection d'intrusions. Comme dans toute approche explicite de la corrélation d'alertes, une bibliothèque de chroniques doit être spécifiée.

Le langage ADeLe présenté dans la section 2.4 fait partie des langages destinés à la corrélation explicite. En effet, les scénarios explicités dans ce langage peuvent ensuite être utilisés pour configurer des SDI corrélant des alertes.

Dans [77], Pouzol et Ducassé proposent un langage de signature déclaratif, *Sutekh*. Les éléments de base de ce langage sont les filtres. Un filtre permet de spécifier une contrainte entre un événement et un terme, un terme étant un ensemble de constante ou une variable. Une signature est en fait une combinaison de filtres construite à par-

```
rule Suspect_scenario(scenario: table[cmnd:string]; i, n, countdown:
integer)
# cette règle surveille la progression dans une suite de commandes
# suspectes i est l'index courant dans le tableau, n est la longueur
# du tableau, countdown est initialisé par le nombre maximum de fois
# que l'on peut observer le scénario. La règle reste active tant que
# countdown n'est pas nul.

if evt = scenario[i]
-> if i<n -> Trigger off for next
Suspect_scenario(scenario,i+1,n,countdown);
i=n -> if countdown > 1
-> Trigger off for next
Suspect_scenario(scenario, 1, n, countdown-1) ;
countdown = 1
-> Alarm(suspect scenario occurs)
fi
fi;
true
-> Trigger off for next Suspect_scenario(scenario, i, n, countdown)
fi
```

FIG. 4.2 – Exemple de règle écrite dans le langage RUSSEL détectant une séquence d'actions enregistrées dans un tableau

tir des opérateurs *then*, *or*, *and* et if_not. Les signatures peuvent aussi inclure des appels à des prédicats logiques externes grâce à l'opérateur *such_that*. Des fonctions externes peuvent être incluses dans une signature grâce à l'opérateur *trigger*. Les auteurs proposent une implantation de leur langage à travers la traduction de signatures écrites en *Sutekh* vers des langages de signature existants. Pour ce faire, une représentation intermédiaire des signatures *Sutekh* est générée. Cette représentation est un graphe état-transition proche de ceux définis par le langage $STAT$. Les auteurs ont implanté la traduction de signatures *Sutekh* en règles $P-Best$ et en règles $RUSSEL$. Ainsi, il est possible de spécifier des signatures pour des SDI existants en utilisant le formalisme de *Sutekh* qui est de plus haut niveau que les langages utilisés pour configurer les SDI. *Sutekh* permet donc de configurer des SDI mais ne se destine pas à la configuration de modules de traitement d'alertes corrélant les alertes en provenance de plusieurs SDI.

Dans [78], les auteurs présentent une approche à l'analyse des logs applicatifs se basant sur la notion de model checking. Les auteurs définissent une logique temporelle du premier ordre ainsi que deux algorithmes permettant d'appliquer la vérification en-ligne ou hors-ligne.

4.1.3 Corrélation semi-explicite

Dans [75], les auteurs argumentent que certaines attaques préparent le système visé pour d'autres attaques. Ils proposent de trouver ces liens entre les alertes élémentaires générées par les SDI. Les auteurs modélisent les actions exécutées par l'attaquant par leur pré-requis et leur conséquence sur le système. Ils proposent d'exprimer ces pré-requis et conséquences par des prédicats logiques. Par exemple, une action consistant à scanner des machines peut permettre de découvrir une vulnérabilité de type dépassement de capacité, ce qui peut être exprimé par le prédicat $UDPVulnerableToBOF(VictimIP, VictimPort)$. Ce prédicat représente la conséquence indirecte du scan et peut être inclus dans les pré-requis d'une attaque se basant sur l'exploitation d'une faille de type dépassement de capacité. En effet, la conséquence directe du scan est que l'attaquant sait que la vulnérabilité existe. Une attaque peut avoir plusieurs pré-requis et plusieurs conséquences. Afin d'exprimer cela, les auteurs proposent d'utiliser les opérateurs de conjonction et de disjonction.

Afin de représenter les alertes, les auteurs introduisent la notion d'hyper-alerte. En fait ils ne proposent pas de modélisation du point de vue de l'attaquant, mais une augmentation de la sémantique des alertes. En effet, les auteurs modélisent des événements détectables, ainsi la modélisation de l'intrusion se fait du point de vue de la détection et non pas du point de vue de l'attaquant. Un type d'hyper-alerte est un triplet $(fait, pre\text{-}requis, consequence)$ où $fait$ est un ensemble de noms d'attributs avec leur domaine associé, *pre-requis* est une combinaison de faits dont les variables sont dans $fait$ et *consequence* est une combinaison de faits dont les variables sont aussi dans $fait$. L'ensemble $fait$ donne le type d'information associé à une alerte et les ensembles de prédicats *pre-requis* et *consequence* expriment respectivement ce qui doit être vérifié pour exécuter l'attaque et les conséquences

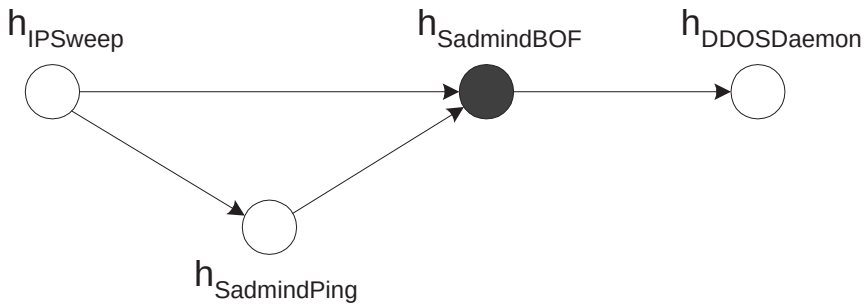

FIG. 4.3 – Exemple de de graphe d'hyper-alertes

possibles de l'attaque sur le système. Une alerte de détection d'intrusions est associée à une instance d'hyper-alerte. Les auteurs précisent qu'une instance d'hyper-alerte, peut être associée à plusieurs alertes générées par des SDI. Cependant aucun détail n'est donné sur le principe du mécanisme de regroupement d'alertes en une seule hyper-alerte.

Les auteurs définissent ensuite la corrélation d'hyper-alertes comme suit. Soient deux hyper-alertes h_1 et h_2. h_1 et h_2 seront considérées comme corrélées si dans les pré-requis de $h2$ il existe un prédicat qu'on retrouve également dans les conséquences de h_1, les variables de ce prédicat ayant les mêmes valeurs dans h_1 et h_2. A partir de cette définition de lien de corrélation entre alertes, les auteurs définissent la notion de graphe de corrélation d'hyper-alertes. Un graphe de corrélation d'hyper-alertes est un DAG (Direct Acyclic Graph) où les nœuds sont des hyper-alertes et les arêtes des liens de corrélation comme définis précédemment. La figure 4.3 représente le graphe de corrélation d'hyper-alertes associé à la détection de quatre attaques et à la formation de quatre hyper-alertes.

Les hyper-alertes composant ce graphe sont les suivantes :

- Attaque $SadmindBufferOverflow =$
  $(VictimIP, VictimPort,$
  $ExistHost(VictimIP) \vee VulnerableSadmind(VictimIP),$
  $GainRootAccess(VictimIP))$.
  Ce type d'hyper alerte décrit une attaque du type dépassement de capacité, son exécution nécessite une vulnérabilité exprimée par le fait $VulnerableSadmind$ $(VictimIP)$ qui exprime que la machine possédant l'adresse IP $VictimIP$ est vulnérable.

- Attaque $SadmindPing =$
  $(VictimIP, VictimPort,$
  $ExistHost(VictimIP),$
  $VulnerableSadmind(VictimIP))$.
  Ce type d'hyper alerte décrit une attaque visant à acquérir un savoir sur le système visé. Une fois cette attaque exécutée avec succès, l'attaquant sait que

la machine visée est vulnérable à une attaque du type dépassement de capacité. Il est important de noter que l'on n'exprime pas explicitement le gain de connaissance pour l'attaquant. Ce gain est représenté par la présence d'un prédicat exprimant l'existence de la vulnérabilité. Les auteurs ne proposent pas de prédicat s'appliquant à une variable représentant le sujet ayant obtenu un gain de connaisances.

– Les attaques $SIPsweep$ et $SDDOSDaemon$ ne sont pas explicitées dans l'article mais nous pouvons préciser ici le principe de ces attaques. L'attaque $SIPsweep$ est du type scan et permet de connaitre les services qui sont actifs sur une machine. Cette attaque permet par exemple de savoir qu'une machine est vulnérable à une attaque de type dépassement de capacité. Les attaques $SadmindBufferOverflow$ et $SDDOSDaemon$ sont corrélées car l'exécution d'une attaque de type dépassement de capacité permet l'exécution d'un code arbitraire et, par exemple, l'installation d'un démon DDOS.

Les auteurs proposent une implantation sur la base d'un moteur de corrélation écrit en Java. Le principe de l'implémentation consiste à pré-traiter les alertes stockées dans une base de données afin de générer les hyper-alertes en instanciant leurs variables. Une fois ces hyper-alertes générées, le problème de trouver les liens de corrélation revient à chercher des chaînes de caractères identiques. Une fois les graphes de corrélation générés, $GraphViz$ est utilisé pour afficher ces graphes. On notera que cette implantation se destine à un usage hors-ligne et ne permet pas la détection de scénarios en temps réel. Cette restriction interdit toute possibilité de réaction en temps réel aux intrusions.

Dans [84] Templeton et Levitt définissent un langage de description d'attaques, Jigsaw, permettant de décrire les attaques de base disponibles pour l'attaquant. Cette description inclut les pré-requis et les conséquences de l'attaque. Les auteurs ne proposent pas de langage pour exprimer les pré-requis et les conséquences d'une attaque. Ils proposent d'exprimer des contraintes sur des types de données encapsulant les informations sur l'événement associé au modèle. La figure 4.4 représente le modèle de l'attaque consistant à établir une connexion RSH en utilisant une adresse forgée.

Les auteurs définissent deux utilisations de ce langage. La première est la découverte de scénarios à partir d'une base d'attaque en identifiant les liens de corrélation entre les modèles. Les auteurs ne définissent aucun algorithme permettant de découvrir ces liens et ne proposent pas d'implantation. La deuxième utilisation possible de cette approche est la reconnaissance de scénarios d'attaques en temps réel. Les auteurs ne proposent pas d'implantation et donnent juste quelques précisions sur la place d'un tel système dans une architecture de détection d'intrusions.

La corrélation semi-explicite a plusieurs avantages. Supposons que nous ayons une base de modèles décrite dans un certain langage et que nous soyons capables de trouver les liens de corrélation entre les modèles. Nous sommes alors capables de générer tous les scénarios réalisables à partir de ces modèles. Ainsi, il est possible de

```
concept RSH_Connection_Spoofing is

      requires
          Trusted_Partner:    TP;
          Service_Active:     SA;
          PreventPacketSend:  PPS;
          extern SeqNumProbe: SNP;
          ForgedPacketSend:   FPS;
      with
          TP.service is RSH,             #- The service in the trust relation is RSH
          PPS.host is TP.trusted,        #- The blocked host is the trusted partner
          FPS.dst.host is TP.trustor,    #- The spoofed packets are sent to the trustor
          SNP.dst.host is TP.trustor,    #- The probed host is the trustor
          FPS.src is [ND.host,PPS.port]  #- claimed source of forged packets is blocked

          SNP.dst is [SA.host,SA.port]   #- The probed host must be running RSH on the
          SA.port is TCP\RSH,            #-   normal port
          SA.service is RSH,             #-

          SNP.dst is FPS.dest            #- probed host must be where forged packets are sent

          active(FPS) during active(PPS)#- forged packets must be sent while DOS is active
      end;

      provides
          push_channel:      PSC;
          remote_execution:  REX;
      with
          PSC.from    <- FPS.true_src;   #- Capability to move code from attacker to RSH serv
          PSC.to      <- FPS.dst;        #-
          PSC.using   <- RSH;            #-

          REX.from    <- FPS.true_src;   #- Capability to execute code on remote host
          REX.to      <- FPS.dst;        #-
          REX.using   <- RSH;            #-
      end;

      action
          true -> report ("RSH Connection Spoofing: TP.hostname")
      end;
end.
```

FIG. 4.4 – Modèle Jigsaw de l'attaque consistant à réaliser une connexion RSH avec une adresse forgée

représenter les variantes d'un scénario connu par exemple. De plus, la maintenance de cette base de modèles est plus simple que celle d'une base de scénarios. En effet, après avoir ajouté un nouveau modèle à la base de modèles la base de scénarios correspondante peut être automatiquement générée.

4.2 Modélisation de l'intrusion

Nous avons présenté les différents types de corrélation ainsi que les langages utilisés dans les approches explicites et semi-explicites. Nous allons maintenant voir quelle approche nous adoptons pour modéliser le processus d'intrusion. Nous commençons par voir comment nous modélisons une intrusion puis nous voyons ensuite comment nous modélisons les actions élémentaires utilisables par l'attaquant. Nous présentons aussi la façon dont nous représentons les objectifs d'un attaquant puis enfin nous terminons par la représentation des propriétés du système attaqué.

4.2.1 L'intrusion en tant que processus de planification

Afin de modéliser le processus d'intrusion, nous nous plaçons du point de vue de l'attaquant. L'attaquant possède un certain savoir sur la machine qu'il vise et a à sa disposition un ensemble d'actions qu'il peut réaliser. Le savoir de l'attaquant peut être nul, ce qui peut avoir un impact direct sur le choix des actions qu'il va réaliser. Plus précisément, un attaquant peut obtenir des informations sur un système informatique en exécutant certaines actions. De manière plus générale le comportement d'un attaquant tout au long du processus d'intrusion peut se découper en deux phases :

- Phase de collecte d'informations : l'attaquant enrichit sa connaissance sur la cible choisie en exécutant certaines actions. Cette phase peut se faire de manière automatique en utilisant par exemple un outil de découverte de vulnérabilités. Ainsi, un balayage des ports d'une machine permet de connaître les services ouverts ; le protocole ICMP peut être utilisé pour découvrir quel système d'exploitation est utilisé.

- Phase d'attaque : une fois que l'attaquant possède suffisamment de connaissances sur le système visé, il peut planifier son attaque en utilisant autant d'actions que nécessaire pour atteindre son objectif d'intrusion. Il est évidemment possible que l'attaquant soit peu expérimenté et utilise des outils d'attaque ou des scripts d'attaque. Dans ce cas, la phase de récupération d'informations peut être absente de la stratégie de l'attaquant.

Cependant un attaquant peut aussi utiliser des attaques ne nécessitant qu'une seule action. Par exemple l'attaque *Winnuke* permet de réaliser un déni de service sur un système fonctionnant sous Windows 95 et Windows NT 3.51/4.0 en envoyant un seul paquet vers la machine cible.

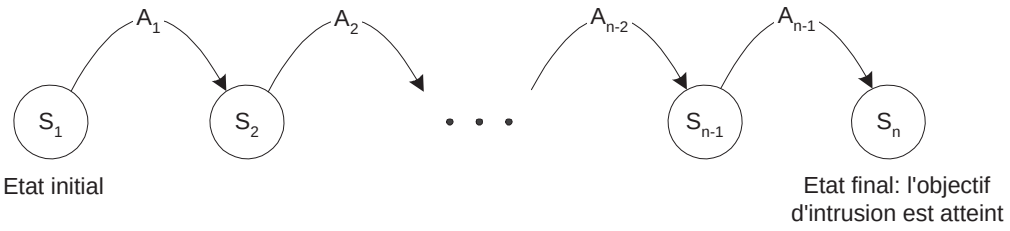

FIG. 4.5 – Modélisation du processus d'intrusion. Les différents états du système sont représentés par les S_i et les actions par les A_i

De manière générale, l'attaquant se fixe un objectif, que nous appelons objectif d'intrusion, qui est représenté par un certain état du système informatique après que l'attaque ait été réalisée. Du point de vue de l'attaquant, l'objectif d'intrusion est la motivation qui justifie ses actions. Du point de vue de la détection, l'objectif d'intrusion est beaucoup plus difficile à définir. En effet, il est difficile de différencier un comportement volontairement malveillant d'un comportement accidentellement intrusif. Afin de lever toute ambiguïté, nous définissons la notion d'objectif d'intrusion :

Définition 4 *Objectif d'intrusion :*
Toute violation de la politique de sécurité est considérée comme étant un objectif d'intrusion.

Par exemple l'obtention d'un accès non autorisé au système avec les droits d'administrateur pour modifier une information constitue un objectif d'intrusion violant la propriété *d'intégrité* du système. Une lecture non autorisée d'un fichier confidentiel constitue un objectif d'intrusion violant la propriété de *confidentialité* du système. Un autre objectif d'intrusion violant la propriété de *disponibilité* du système peut être constitué par la réalisation d'un déni de service sur une machine, par exemple en utilisant l'attaque *Winnuke*.

Une fois que l'attaquant a défini son ou ses objectifs d'intrusion, il détermine une stratégie représentée par un ensemble d'actions à exécuter. L'exécution de ces actions, organisées en un scénario d'intrusion, permet de changer l'état du système attaqué d'un état initial sain, où la politique de sécurité est respectée, en un état final où l'objectif d'intrusion est atteint et la politique de sécurité violée (figure 4.5). Il est à noter que notre schéma peut laisser supposer que les actions doivent être exécutées séquentiellement, ce qui n'est pas obligatoire.

Avant de présenter comment nous représentons les actions utilisées par l'attaquant, il est nécessaire de définir précisément les termes *action* et *attaque*. En effet, nous pouvons distinguer trois types d'actions exécutées par l'attaquant :

– Les actions normales : ce sont des actions qui modifient l'état du système sans violer la politique de sécurité. Ce sont donc des actions autorisées par la politique de sécurité.

- Les actions suspectes : ce sont des actions dont les modifications sur l'état du système ne violent pas la politique de sécurité mais qui sont corrélées à une action malveillante. Nous définissons informellement le terme de corrélation ci-dessous.
- Les actions malveillantes : ce sont des actions dont l'exécution permet d'atteindre directement un objectif d'intrusion.

Définition 5 *Corrélation d'actions (définition informelle) :*
Une action A_1 est corrélée à une action A_2 si son exécution peut permettre à l'attaquant d'exécuter A_2. On dit aussi que A_1 a une **influence positive** *sur A_2.*

Ce concept de corrélation est formalisé plus loin. Les actions suspectes et malveillantes sont aussi appelées attaques. Cela peut sembler être une définition assez faible du terme « attaque » car nous désignons les actions suspectes par le même terme, cependant ce type d'action constitue une grande partie des alertes reçues.

Nous allons maintenant exposer comment nous modélisons les actions utilisées par l'attaquant.

4.2.2 Représentation des actions

Dans le domaine de la planification, les actions sont souvent représentées par leur pré-condition et leur post-condition. La pré-condition d'une action représente l'ensemble des conditions que l'état du système doit satisfaire pour que l'action puisse être exécutée. La post-condition d'une action représente ses effets sur l'état du système ou l'état de connaissance de l'attaquant.

Nous représentons une action en décrivant trois champs : son nom, sa pré-condition et sa post-condition. Nous utilisons pour cela un sous-ensemble du langage LAMBDA présenté dans la section 2.4. Plus précisément, le nom de l'action est une expression fonctionnelle comprenant le nom de l'action et ses différents paramètres. La pré-condition est une expression logique exprimant les conditions logiques *minimales* permettant l'exécution de l'action. La post-condition est une expression logique représentant les effets *minimaux* de l'exécution de l'action sur l'état du système ou l'état de connaissance de l'attaquant.

Par rapport au langage LAMBDA (voir section 2.4), voici les restrictions que nous appliquons :

- Restrictions sur le langage L_1 : nous n'utilisons pas l'opérateur de disjonction $\vee$, ceci afin de minimiser les phénomènes d'explosion combinatoire. L'opérateur de négation $\neg$ ne s'applique qu'à des prédicats et non pas à des conjonctions de prédicats.
- Champs LAMBDA omis : les champs *scenario* et $verification$.

Ainsi nous donnons la définition d'un modèle d'action :

Définition 6 *Modélisation des actions*
Une action A est modélisée par trois champs :

- *$Name(Param_1, Param_2, ..., Param_n)$: expression fonctionnelle représentant le nom de l'action ainsi que ses paramètres.*
- *Pré-condition : conjonction de prédicats que le système doit satisfaire pour que l'action puisse s'effectuer.*
- *Post condition : conjonction de prédicats exprimant les effets de l'action sur le système ou sur les connaissances de l'attaquant.*
- *Détection : conjonction de prédicats indiquant comment instancier les variables du modèle.*

La figure 4.6 représente quelques exemples de modèles d'actions utilisées dans différents scénarios d'attaque. Les virgules séparant les prédicats dans la pré-condition et la post-condition représentent l'opérateur conjonctif. Ces scénarios sont explicités dans la section 6.2.1. Afin de représenter un gain de connaissance de l'attaquant sur le système, nous définissons la modalité *knows*. Cette modalité s'utilise de la manière suivante : $knows(S, expr)$ signifie que l'agent S sait que $expr$ est vraie. Prenons l'exemple du prédicat $knows(User, use\_service(Address, fingerd))$. Ce prédicat signifie que l'agent $User$ sait que $use\_service(Address, fingerd)$ est vraie, c'est-à-dire que la machine possédant l'adresse $Address$ héberge le service $fingerd$. La modalité *knows* ne s'applique que sur des prédicats ou des négations de prédicats. Le champ *Detection* exprime d'une part comment associer un modèle et une alerte et d'autre part comment instancier les variables. Ainsi, *classification(Alert,'mount')* exprime que si l'événement associé à une alerte est *mount*, ceci étant déterminé par l'ensemble $\mathcal{E}_{vt}$ défini dans le chapitre précédent et le dictionnaire d'événement, alors le modèle d'action $mount(User, Address, Partition)$ doit être associé à cette alerte et instancié. Le prédicat $source(Alert, User)$ spécifie que la valeur de l'adresse source présente dans l'alerte doit être utilisée pour instancier la variable $User$. De même, $target(Alert, Address)$ spécifie d'instancier la variable $Address$ avec l'adresse destination de l'alerte spécifiée dans la classe $Target$ pour une alerte IDMEF.

Le premier modèle $mount(User, Address, Partition)$ modélise l'exécution du montage d'une partition UNIX (ou Linux) à distance. Les variables $User$, $Address$ et $Partition$ désignent respectivement l'utilisateur exécutant l'action, l'adresse de la machine sur laquelle la partition à monter réside et enfin la partition à monter. L'exécution de cette action nécessite que la partition à monter soit montée sur l'ordinateur distant et que l'utilisateur ait un accès à distance sur cet ordinateur. L'effet de cette action est que l'utilisateur peut accéder à la partition. Le deuxième modèle $rpcinfo(User, Address)$ représente une action permettant à l'utilisateur de récupérer des informations sur les services démarrés sur une machine. Les variables $User$ et $Address$ désignent respectivement l'utilisateur exécutant l'action et l'adresse de la machine dont on veut connaître les services. La modalité *knows* présente dans la post-condition exprime le gain de connaissances sur la machine visée. En général, ces deux premiers modèles d'action ne représentent pas des actions malveillantes mais des actions suspectes. En effet, la première action permet à l'attaquant d'obtenir un accès à un support de fichier pour poursuivre son attaque, cette action faisant partie d'un scénario que nous présentons plus loin. La deuxième action modélisée présente une première étape dans le même scénario, cette étape fait partie de la phase de

Action $mount(User, Address, Partition)$
Pre : $remote\_access(User, Address)$,
$mounted\_partition(Address, Partition)$
Post : $can\_access(User, Partition)$
Detection : *classification(Alert,'mount')*,
$source(Alert, User)$,
$target(Alert, Address)$,
$target\_partition(Alert, Partition)$

Action $rpcinfo(User, Address)$
Pre : $remote\_access(User, Address)$,
$use\_service(Address, mountd)$
Post : $knows(User, use\_service(Address, mountd))$
Detection : *classification(Alert,'rpcinfo')*,
$source(Alert, User)$,
$target(Alert, Address)$

Action $syn\_flood(User, Address)$
Pre : $remote\_access(User, Address)$,
Post : $dos(Address)$
Detection : *classification(Alert,'SYN-flooding')*,
$source(Alert, User)$,
$target(Alert, Address)$

FIG. 4.6 – Exemples de modèles d'actions

récupération des informations sur le système visé.

Le troisième modèle d'action $syn\_flood(User, Address)$ représente l'exécution d'une attaque de type syn-flooding. Les variables $User$ et $Address$ désignent respectivement l'utilisateur exécutant l'action et l'adresse de la machine visée par l'attaque. Cette action est malveillante car elle permet d'atteindre un objectif d'intrusion si la machine visée par l'attaque et rendue indisponible doit satisfaire la propriété de disponibilité conformément à la politique de sécurité. Il est à noter que pour que l'action soit effective, il est nécessaire que la machine cible soit vulnérable à une attaque de ce type. Cette condition est exprimée par le prédicat $vulnerable(Address, syn\_flooding)$. Cet exemple introduit la nécessité de modéliser les caractéristiques du système surveillé. Nous verrons plus loin dans la section 4.2.4 comment nous représentons cette connaissance.

Lors de la mise en œuvre du processus de détection, nous sommes incapables d'observer directement l'exécution des actions par l'attaquant. En effet, du point de vue de la détection d'intrusions nous ne pouvons observer que les manifestations de ces actions par l'occurrence de certains événements qui correspondent aux alertes générées par les sondes ou enregistrés dans les logs applicatifs. Lorsque nous sommes capables d'associer un modèle d'action à un événement détecté par l'ensemble des SDI déployés, nous créons une instance d'action correspondant au modèle. Cette instance d'action possède les mêmes variables que le modèle mais instanciées. De plus nous lui associons une date, cette date étant celle de la détection de l'événement. Ainsi, nous modélisons une instance d'action de la manière suivante :

Définition 7 *Modélisation des instances d'actions :*
Une instance d'action A est modélisée par quatre champs :
- $Name(Param_1, Param_2, ..., Param_n)$ *: expression fonctionnelle représentant le nom de l'action ainsi que ses paramètres instanciés.*
- *DetectTime : la date à laquelle l'action a été détectée.*
- *Pré-condition : conjonction de prédicats instanciés que le système doit satisfaire pour que l'action puisse s'effectuer.*
- *Post condition : conjonction de prédicats instanciés exprimant les effets de l'action sur le système.*

Le champ *Détection* n'apparaît pas dans une instance d'action car ses variables sont instanciées. Dans la suite nous désignerons par $Pre(A)$ la pré-condition d'une action ou d'une instance d'action et par $Post(A)$ la post-condition.

Nous avons présenté la façon dont nous représentons les actions, un ensemble d'actions formant un scénario menant éventuellement à un objectif d'intrusion. Nous allons maintenant voir comment nous représentons les objectifs d'intrusion et définir formellement la notion de scénario d'intrusion.

4.2.3 Représentation des objectifs d'intrusion

Nous avons donné précédement une définition informelle d'un objectif d'intrusion en précisant que cela correspond à une violation de la politique de sécurité. Ici, nous proposons de représenter un objectif par une condition sur l'état du système, cette condition correspondant à une violation de la politique de sécurité. Cette condition est exprimée de la même manière que la pré-condition d'une action et avec les mêmes restrictions par rapport au langage L_1 de LAMBDA.

Définition 8 *Modélisation des objectifs d'intrusion*
Un objectif d'intrusion O est modélisé par deux champs :
- $Name(Param_1, Param_2, ..., Param_n)$ *: expression fonctionnelle représentant le nom de l'objectif ainsi que ses paramètres instanciés.*
- *StateCondition : conjonction de prédicats que le système satisfait.*

La figure 4.7 donne trois exemples de modèles d'objectifs d'intrusion. Ces trois modèles correspondent aux exemples donnés en 4.2.1 illustrant respectivement la violation des propriétés d'intégrité, de confidentialité et de disponibilité.

Un modèle d'objectif d'intrusion est instancié une fois que la dernière action ayant permis de satisfaire la condition sur l'état du système a été observée. La date à laquelle l'objectif d'intrusion est atteint correspond donc à la date de la dernière instance d'action permettant de satisfaire tous les prédicats de la condition associée à l'objectif. De la même manière que pour les instances d'action, nous modélisons les instances d'objectif d'intrusion en ajoutant un champ donnant la date à laquelle l'objectif est atteint.

Définition 9 *Modélisation des instances d'objectifs d'intrusion*
Une instance d'objectif d'intrusion O est modélisé par trois champs :

Intrusion_Objective $illegal\_root\_access(Agent, Host)$ State_Condition : $root\_access(Agent, Host)$, not$(authorized(Agent, root, Host))$
Intrusion_Objective $illegal\_file\_access(File)$ State_Condition : $read\_access(Agent, File)$, not$(authorized(Agent, read, File))$
Intrusion_Objective $DOS\_on\_DNS(Host)$ State_Condition : $dns\_server(Host), dos(Host)$

FIG. 4.7 – Exemples d'objectifs d'intrusion

- $Name(Param_1, Param_2, ..., Param_n)$ *: expression fonctionnelle représentant le nom de l'objectif ainsi que ses paramètres.*
- *AchieveTime : la date à laquelle l'objectif d'intrusion a été atteint.*
- *StateCondition : conjonction de prédicats que le système satisfait une fois l'objectif atteint.*

A partir des définitions précédentes d'instance d'action et d'objectif d'intrusion, nous proposons une définition de scénario d'intrusion :

Définition 10 *Scénario d'intrusion*

Un scénario d'intrusion est une séquence $S(A_1, A_2, ..., A_n, O)$ *ou les* A_i *sont des instances d'actions et* O *est une instance d'objectif d'intrusion telles que :*

- $\forall i, j \in \{1, ..., n\}$*, si* $i > j$ *alors* $Detectime(A_i) \geq Detectime(A_j)$
- $\forall i \in \{2, ..., n\}$, $\exists j < i$ *tel que* A_j *a une influence positive sur* A_i*.*
- A_n *est une action malveillante pour* O

Il est à noter que d'autres actions (autres que A_n) du scénario S peuvent aussi avoir une influence positive sur O. Par influence positive d'une action A sur une action B, nous désignons le fait que la réalisation de A permet de satisfaire certains prédicats de la pré-condition de l'action B. Un scénario peut comporter plusieurs actions initiales correspondant aux premières actions exécutées par l'attaquant. Nous définissons ces actions initiales de la manière suivante :

Définition 11 *Action initiale*

Une action A est une action initiale si sa pré-condition est égale à vrai ou si tous les prédicats de la pré-condition sont satisfaits par l'état initial du système, c'est-à-dire l'état du système au début du scénario.

Nous avons montré comment nous modélisons le processus d'intrusion à travers la modélisation des actions exécutables par l'attaquant ainsi que la modélisation des objectifs d'intrusion. Nous avons vu que l'expression des pré-conditions des actions, ainsi que l'expression des conditions sur l'état du système des objectifs d'intrusion, font intervenir une certaine connaissance sur les propriétés du système surveillé.

Nous avons aussi vu que l'objectif d'intrusion DOS_on_DNS de la figure 4.7 utilise le prédicat $dns\_server(Host)$ indiquant que la machine $Host$ est un serveur DNS. Nous abordons le problème de la représentation de ces propriétés dans la section suivante.

4.2.4 Représentation de la connaissance sur le domaine ou l'état du système

La connaissance sur le domaine, ou état du système, est représentée par K et contient les informations disponibles sur le système. Cette connaissance est représentée par un ensemble de prédicats. Par exemple, la connaissance K du domaine peut être égale à $\{file(secret\_file), printer(ppt), physical\_access(Agent, ppt)\}$, qui veut dire que $secret\_file$ est un fichier, ppt est une imprimante et que $Agent$ peut accéder à l'imprimante ppt. Ces données pourraient par exemple être contenues dans une base de données s'appuyant sur le modèle formel $M2D2$ défini dans [71].

La collecte des informations sur l'état du système et la modification de cette connaissance se fait à travers la réception des alertes de détection d'intrusions. Cependant cette connaissance ne se limite pas aux informations fournies par les alertes mais s'étend aux informations que nous qualifions de topologiques. Ces informations correspondent aux caractéristiques des entités du réseau surveillé. Comme indiqué dans [71], il est nécessaire de corréler les informations apportées par les alertes avec les propriétés du matériel ou du logiciel visé par les actions. En effet, la réussite de certaines attaques, et plus généralement les effets de l'exécution d'une action, dépendent de la configuration du système. Par exemple, l'attaque *winnuke* ne peut être un succès que si le système d'exploitation de la machine visée est Windows 98 ou Windows NT 3.51/4.0. Ce type d'information peut être collectée en partie par des outils de découverte de topologie de réseau et de vulnérabilité comme Nessus ([30]), Nmap ([44]) ou Ntop ([32]).

La connaissance sur le domaine comprend aussi certaines propriétés pouvant s'exprimer par des règles permettant de déduire des informations à partir d'autres informations. Nous représentons ce type de règles par une pré-condition et une post-condition. La pré-condition représente une certaine connaissance sur le système et la post-condition représente les conclusions que l'on dérive de cette connaissance. Dans ce cas, nous ne modélisons pas le changement d'état provoqué par l'exécution d'une action ; la pré-condition et la post-condition de ces règles sont évaluées sur le même état du système. Deux exemples simples sont représentés sur la figure 4.8. La première règle spécifie que le possesseur d'un fichier possède les droits d'écriture et de lecture sur ce fichier. Notons l'introduction du prédicat *authorized* qui spécifie l'autorisation possédée par un agent sur un objet du système. Par exemple, l'expression $authorized(agent_1, write, file_1)$ représente le fait que l'agent $agent_1$ possède les droits d'écriture sur le fichier $file_1$.

La deuxième règle spécifie que si un fichier est supprimé, il n'y a plus de propriétaire pour ce fichier et donc que chaque utilisateur se voit privé de ses droits d'accès en lecture et en écriture.

Domain_rule $owner\_right(File)$ Pre : $owner(Agent, File)$ Post : $authorized(Agent, read, File), authorized(Agent, write, File)$
Domain_rule $remove\_right(File)$ Pre : not $file(File)$ Post : not $(owner(Agent, File))$, not $(authorized(Agent, read, File))$, not $(authorized(Agent, write, File))$

FIG. 4.8 – Exemples de règles définies sur le domaine

4.3 Détection de scénario d'intrusion

Dans la section précédente, nous avons présenté la modélisation adoptée pour représenter les actions exécutées par l'attaquant ainsi que ses objectifs. Nous allons maintenant présenter comment nous utilisons ces modèles pour détecter des ensembles d'actions organisées en scénario d'intrusion. Nous présentons tout d'abord formellement la notion de corrélation d'action définie informellement dans la définition 5. Ensuite, nous verrons comment mettre en œuvre cette notion dans un processus de détection.

4.3.1 La notion de corrélation

La notion de corrélation s'applique à la fois aux actions et aux objectifs d'intrusion. Nous définissons tout d'abord la notion de corrélation entre deux expressions logiques.

Soient E et F deux expressions logiques [1] ayant la forme suivante :

- $E = expr_{E_1}, expr_{E_2}, ..., expr_{E_m}$
- $F = expr_{F_1}, expr_{F_2}, ..., expr_{F_n}$

où chaque $expr_{E_i}$ (resp $expr_{F_j}$) est un prédicat ou la négation d'un prédicat, c'est-à-dire que $expr_{E_i}$ (resp $expr_{F_j}$) doit avoir une des formes suivantes :

- $expr_{E_i} = pred$
- $expr_{E_i} =$ `not` $(pred)$

où *pred* est un prédicat. La définition suivante introduit la corrélation qui se base sur la notion d'unification de la logique du permier ordre.

Définition 12 *Corrélation*

Les expressions E et F sont corrélées *s'il existe i dans $\{1, ..., m\}$ et j dans $\{1, ..., n\}$ tels que $expr_{E_i}$ et $expr_{F_j}$ sont unifiables par un plus grand unifieur (pgu) Θ.*

[1] Nous considérons que ces deux expressions n'incluent pas de disjonctions. Cette restriction nous permet de simplifier la définition de la corrélation. La généralisation de la notion de corrélation afin de considérer les disjonctions représente un travail qui reste à faire.

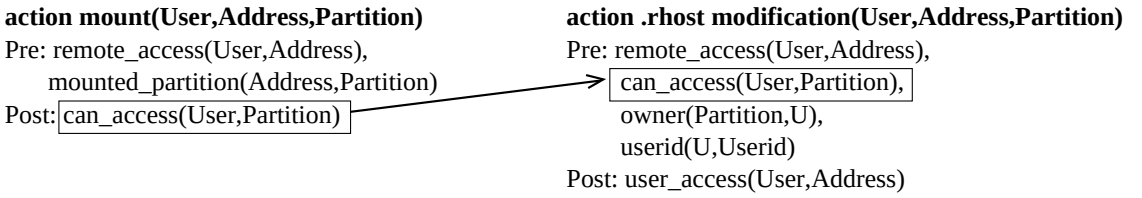

FIG. 4.9 – Exemple d'influence positive entre deux actions

On dit qu'un ensemble d'expressions $\mathcal{F} = f_1, f_2, ..., f_n$ est unifiable s'il existe une substitution S telle que $f_1.S = f_2.S = ... = f_n.S$. Le plus grand unifieur (pgu) d'un ensemble d'expressions $\mathcal{F}$ est une substitution S qui unifie $\mathcal{F}$ et pour toute autre substitution T unifiant $\mathcal{F}$, il existe une substitution V telle que $S.V = T$.

Nous allons maintenant utiliser cette notion pour définir dans un premier temps la notion de corrélation entre attaques puis nous définirons la notion de corrélation avec un objectif d'intrusion.

4.3.2 Corrélation d'actions

Définition 13 *Corrélation directe d'actions ou influence positive*

Une action A a une influence positive *sur une action B si la post condition de A et la pré-condition de B sont corrélées en utilisant la définition 12. On dit que A est directement corrélée à B*

La figure 4.9 illustre un exemple de corrélation directe d'attaques où l'action *mount* peut avoir une influence positive sur l'action *.rhost modification*. *mount* est directement corrélée à *.rhost modification*.

La notion de corrélation directe n'est cependant pas suffisante pour représenter l'influence positive que l'exécution d'une action peut avoir sur la réalisation d'une autre action. En effet, un ensemble de déductions faites à partir d'un sous-ensemble des règles définies sur le domaine peut permettre de conclure à une influence positive *indirecte* d'une action A sur une action B. Ainsi, nous définissons la corrélation indirecte entre actions :

Définition 14 *Corrélation indirecte d'actions ou influence positive indirecte*

Une action A a une influence positive indirecte *sur une action B par l'intermédiaire des règles sur le domaine $R_1, ..., R_n$ si les conditions suivantes sont satisfaites :*

- *la post-condition de A et la pré-condition de R_1 sont corrélées en utilisant la définition 12.*
- *pour chaque j dans $[1, n-1]$, la post-condition de R_j et la pré-condition de R_{j+1} sont corrélées en utilisant la définition 12.*
- *la post-condition de R_n et la pré-condition de B sont corrélées en utilisant la définition 12.*

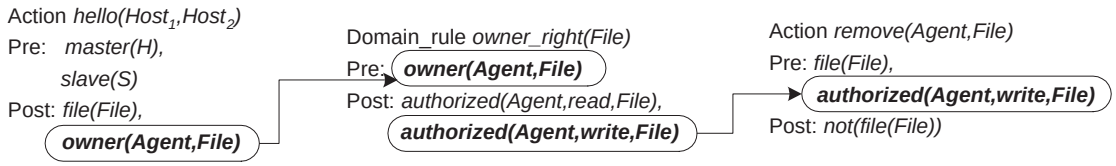

FIG. 4.10 – Exemple de corrélation indirecte

On dit que A est indirectement corrélée avec B

La figure 4.10 présente un exemple de corrélation indirecte entre deux attaques.

4.3.3 Corrélation sur un objectif d'intrusion

Nous avons défini plus haut de manière informelle la notion d'action malveillante en spécifiant que l'exécution d'une telle action permet d'atteindre directement un objectif d'intrusion. Nous définissons ici formellement ce qu'est une action malveillante à travers la notion de corrélation entre une action et un objectif d'intrusion.

Définition 15 *Action malveillante*
Une action A est une action malveillante pour l'objectif d'intrusion O si la post condition de A et la condition sur l'état du système de O sont corrélées en utilisant la définition 12.

Dans les sections précédentes, nous avons vu comment modéliser des actions, des objectifs d'intrusion et des règles sur le domaine représentant certaines propriétés du système. Nous avons ensuite défini la notion de corrélation entre deux actions et entre une action et un objectif d'intrusion. Nous allons maintenant voir comment utiliser ces définition pour mettre en œuvre un processus de détection de scénarios.

4.3.4 Règles de corrélation

Les définitions données précédemment concernent la corrélation d'instances d'actions et d'objectifs d'intrusion. Nous avons aussi précisé que du point de vue de la détection nous n'observons pas directement les actions mais les alertes de détection d'intrusions correspondant à l'exécution de ces actions. Afin d'associer correctement une alerte à l'action qu'elle représente, nous utilisons le dictionnaire et l'ensemble $\mathcal{E}_{vt}$ défini dans le chapitre précédent.

Afin de corréler deux alertes correspondant à deux actions corrélées, nous définissons des règles de corrélation. Considérons l'ensemble des modèles d'actions et d'objectifs d'intrusion. D'après la définition 13, pour chaque couple d'actions corrélées et pour chaque couple action-objectif d'intrusion il existe un plus grand unifieur précisant comment les variables des modèles doivent s'unifier pour que deux instances soient corrélées.

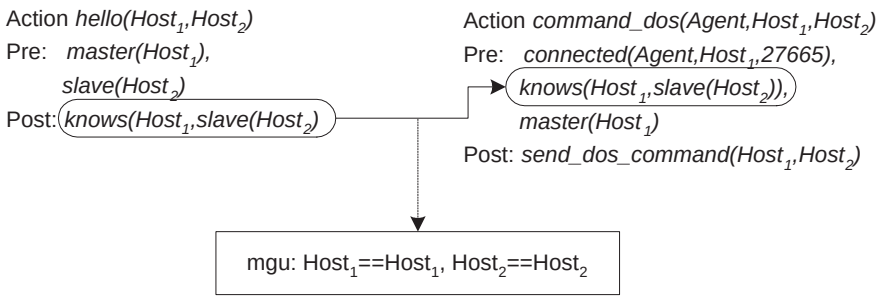

FIG. 4.11 – Exemple de règle de corrélation

L'ensemble de ces unifieurs représente ce que nous appelons l'ensemble des règles de corrélation. Ces règles de corrélation sont utilisées en temps réel pour vérifier si une nouvelle alerte peut être corrélée avec d'autres alertes déjà reçues. Elles sont automatiquement générées à partir de l'ensemble des modèles d'actions, des modèles d'objectifs et des règles sur le domaine.

La figure 4.11 donne un exemple de règle de corrélation générée à partir de deux modèles d'action corrélés. Ces deux modèles font partie de la modélisation de l'attaque par un outil de déni de service distribué, *Trinoo*. L'unifieur fait intervenir certaines variables des deux modèles et les deux égalités doivent être satisfaites par les instances de modèle.

Ainsi, l'ensemble des règles de corrélation nous permettent de détecter des ensembles d'alertes corrélées formant des scénarios d'intrusion. Cependant ce mode de détection ne permet que de constater l'évolution d'un scénario dans un processus de détection en temps réel. Le traitement hors-ligne d'un ensemble d'alertes permet de découvrir un ensemble de scénarios correspondant à une activité intrusive. Ce traitement hors-ligne permet éventuellement de trouver un problème de configuration ou une faille de sécurité dans le système mais ne permet pas bien sûr d'empêcher l'intrusion. Il nous reste à définir d'autres mécanismes permettant de satisfaire les exigences que nous avons présentées dans l'introduction, à savoir être en mesure d'anticiper les intentions de l'attaquant et de réagir afin de contrecarrer l'avancement d'un scénario. Dans la prochaine section, nous présentons la notion de génération d'hypothèse visant à satisfaire le besoin d'anticipation. Le problème de la réaction est traité dans le chapitre 5.

4.4 La génération d'hypothèses

La notion de génération d'hypothèses que nous allons présenter nous permet de répondre à deux problèmes : la gestion des actions non observées, ou faux négatifs, et l'anticipation des intentions de l'attaquant ou reconnaissance d'intention.

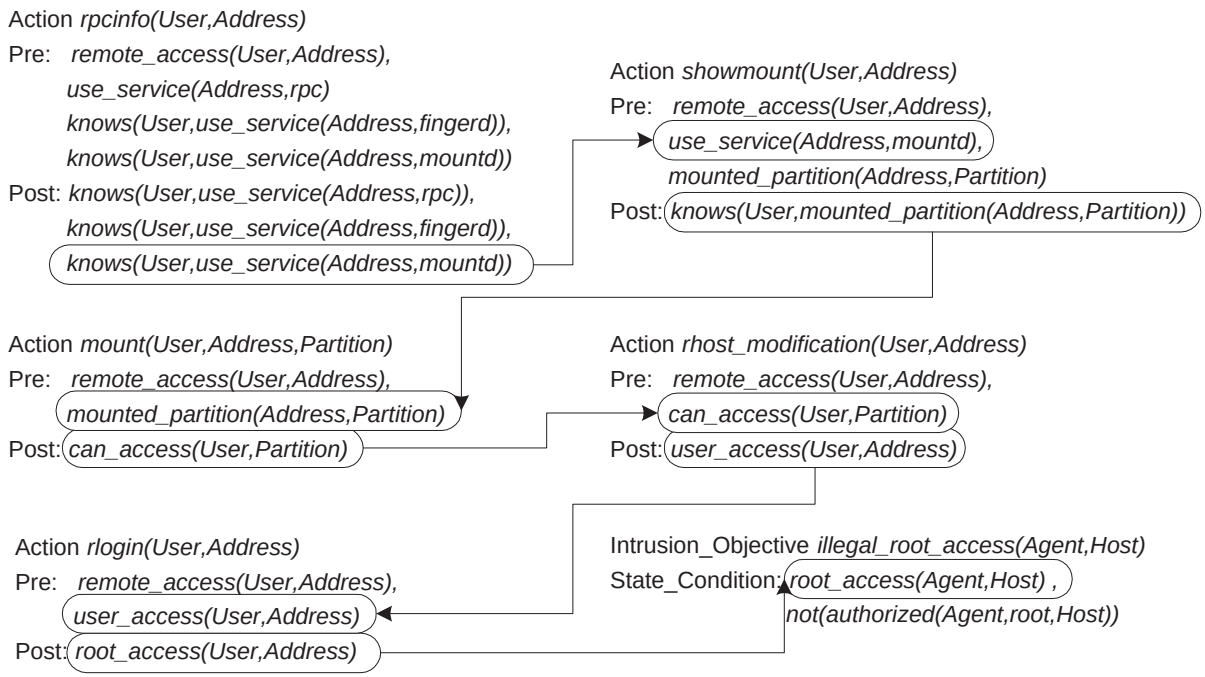

FIG. 4.12 – Liens de corrélation existants dans le scénario d'accès illégal à un système

4.4.1 Gestion des faux négatifs

Comme nous l'avons précisé précédemment, du point de vue de la détection nous ne pouvons pas observer directement les actions exécutées par l'attaquant mais seulement les événements détectés par les SDI. Ainsi, nous pouvons distinguer deux cas de faux négatifs :

– Action non observable : ce type d'action n'est pas détectable de par sa nature. Une action non observable est une action que les SDI ne peuvent pas détecter à cause de leur placement dans le réseau surveillé. Un exemple est l'action *.rhost* du scénario que nous présentons dans la section 6.2.1 consacré à l'expérimentation. La figure 4.12 représente comment les modèles des actions de ce scénario sont corrélées et permettent d'atteindre un objectif d'intrusion violant la propriété d'intégrité du système.
L'action consistant à modifier le fichier.$rhost$ est ici modélisée par l'action $rhost\_modification$ et correspond à une étape du scénario exécutée localement sur l'ordinateur de l'attaquant. Elle est donc indétectable par des sondes qui surveillent la cible. Lors de la tentative de détection d'un tel scénario, nous ne recevons que quatre alertes[2] correspondant à l'exécution des actions $rpcinfo$, $showmount$, $mount$ et $rlogin$. Afin de détecter complètement le scénario, il nous faut définir un mécanisme permettant de prendre en compte l'action non

[2]Nous supposons ici que les alertes produites par les SDI distribués dans le système sont agrégées et fusionnées afin d'obtenir une alerte globale par instance d'action.

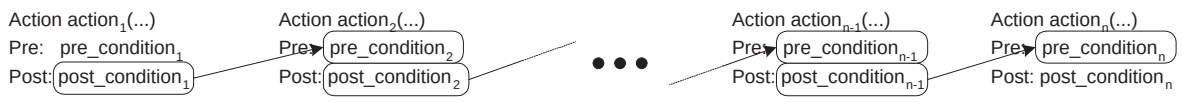

FIG. 4.13 – Ensemble de modèles corrélés

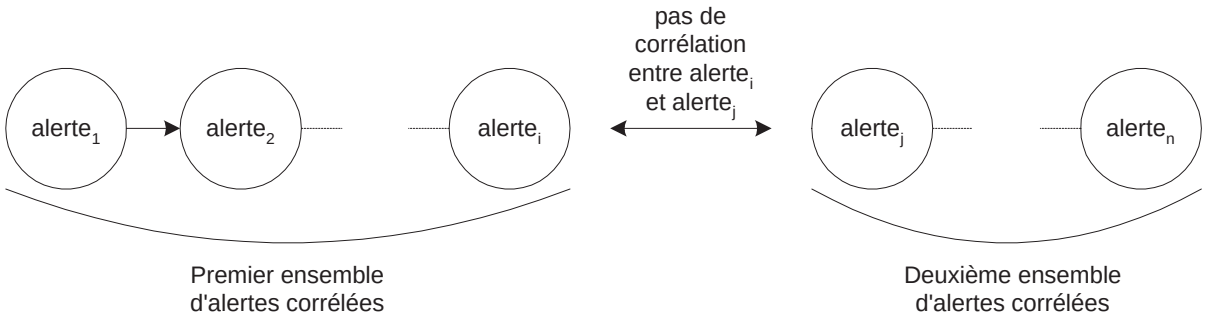

FIG. 4.14 – Deux ensembles d'alertes corrélées faisant partie du même scénario

observée $rhost\_modification$.

– Action non détectée : il se peut qu'un SDI n'émette pas d'alerte lors de l'exécution d'une action par l'attaquant, bien que les manifestations de cette action soient observables sur le système attaqué. En effet, la base de signature d'un SDI peut ne pas contenir la signature permettant de détecter l'événement.

Afin de pallier ce problème, nous proposons le mécanisme de génération d'hypothèses qui consiste à générer les hypothèses minimales permettant d'expliquer les observations, c'est-à-dire les alertes déjà reçues. Considérons un ensemble de modèles d'actions formant une chaîne d'actions corrélées (figure 4.13). Ce modèles correspondent au scénario présenté dans la section 6.2.1.

Supposons que nous observons m instances des n modèles sous la forme de m alertes, m étant strictement inférieur à n. Supposons que les actions non détectées forment une chaîne de $n-m$ actions corrélées. Nous supposons que nous avons deux ensembles d'alertes corrélées E_1 et E_2 appartenant en fait au même scénario mais apparemment indépendantes (figure 4.14). La génération d'hypothèses consiste à explorer les règles de corrélation afin de trouver une chaîne de modèles corrélés permettant de relier deux modèles d'action. Dans notre exemple, la chaîne de modèles corrélés est constituée par l'ensemble $Action_{i+1}, ..., Action_{j-1}$. Une fois qu'une ou plusieurs chaînes de modèles d'action corrélés ont été trouvées, il faut encore vérifier si l'instanciation de ces ensembles permet de réunir E_1 et E_2 dans un même scénario. Dans notre exemple nous considérons qu'un seul ensemble de modèles $Action_{i+1}, ...,$ $Action_{j-1}$ a été trouvé pour corréler E_1 et E_2.

Afin de vérifier si E_1 et E_2 font partie du même scénario, nous devons instancier le modèle $Action_{i+1}$ de façon à ce que son instance, représentée par une alerte dite "virtuelle", soit corrélée avec $alerte_i$. Ensuite nous instancions les modèles $Action_{i+2}$

jusqu'à $Action_{j-1}$ pour vérifier si $alerte_{j-1}$ est corrélée à $alerte_j$. Si c'est le cas, nous avons trouvé une chaîne d'alertes virtuelles formant un ensemble d'hypothèses permettant d'expliquer l'observation des deux ensembles E_1 et E_2 et permettant de conclure que ces alertes font partie du même scénario.

Ce mécanisme a l'avantage de permettre la gestion des faux négatifs. En revanche, il peut générer un nombre important d'hypothèses. Dans notre implantation nous fixons le nombre maximal d'hypothèses générées afin de limiter les calculs effectués. D'autre part l'algorithme peut identifier plusieurs chemins d'alertes virtuelles susceptibles de relier deux alertes. Pour l'instant nous ne pouvons pas affirmer lequel de ces chemins est le plus plausible. Nous proposons une approche permettant de répondre à cette question plus loin dans la section 4.5.

4.4.2 Reconnaissance des intentions de l'attaquant

Nous désignons par reconnaissance d'intentions le mécanisme permettant, à partir des observations représentées par les alertes de détection d'intrusions, de déterminer plusieurs inconnues :

- Les actions futures possibles pouvant être exécutées par l'attaquant : à partir des scénarios d'intrusion partiels observés, nous voulons prévoir les évolutions possibles de ces scénarios.
- Les objectifs poursuivis par l'attaquant : quels sont les objectifs d'intrusion atteignables étant donnés les scénarios d'intrusion partiels identifiés ?

Afin de répondre à ces questions, nous utilisons le même mécanisme que celui présenté dans le cas de la gestion de faux négatifs. Le principe est d'explorer l'ensemble des règles de corrélation afin de trouver un ensemble de modèles corrélés avec le modèle associé à la dernière action d'un scénario partiel (figure 4.15).

Une fois les modèles trouvés, il reste à les instancier sous la forme d'alertes virtuelles. Cette instanciation se fait à partir des valeurs des variables associées aux alertes déjà reçues.

Lors de l'exploration des règles de corrélation, nous pouvons trouver un modèle d'action malveillante corrélée à un objectif d'intrusion. Dans ce cas, l'objectif d'intrusion atteint par la chaîne de modèles corrélés représente un objectif d'intrusion potentiellement visé par l'attaquant. Dans notre implantation, le nombre d'hypothèses pouvant être généré est limité afin d'éviter une explosion combinatoire et afin de pouvoir fonctionner en temps réel.

Tout comme dans le cas de la gestion des faux négatifs, il est possible de trouver un ensemble de sous-ensembles de modèles d'action corrélés, chaque suite d'action pouvant être corrélée à un objectif d'intrusion. Dans un tel cas, le diagnostic remonté à l'administrateur n'est pas suffisamment informatif car une liste de scénarios possibles lui est présentée, sans aucune information lui permettant d'identifier le scénario le plus plausible. Nous proposons une solution à ce problème dans la section suivante.

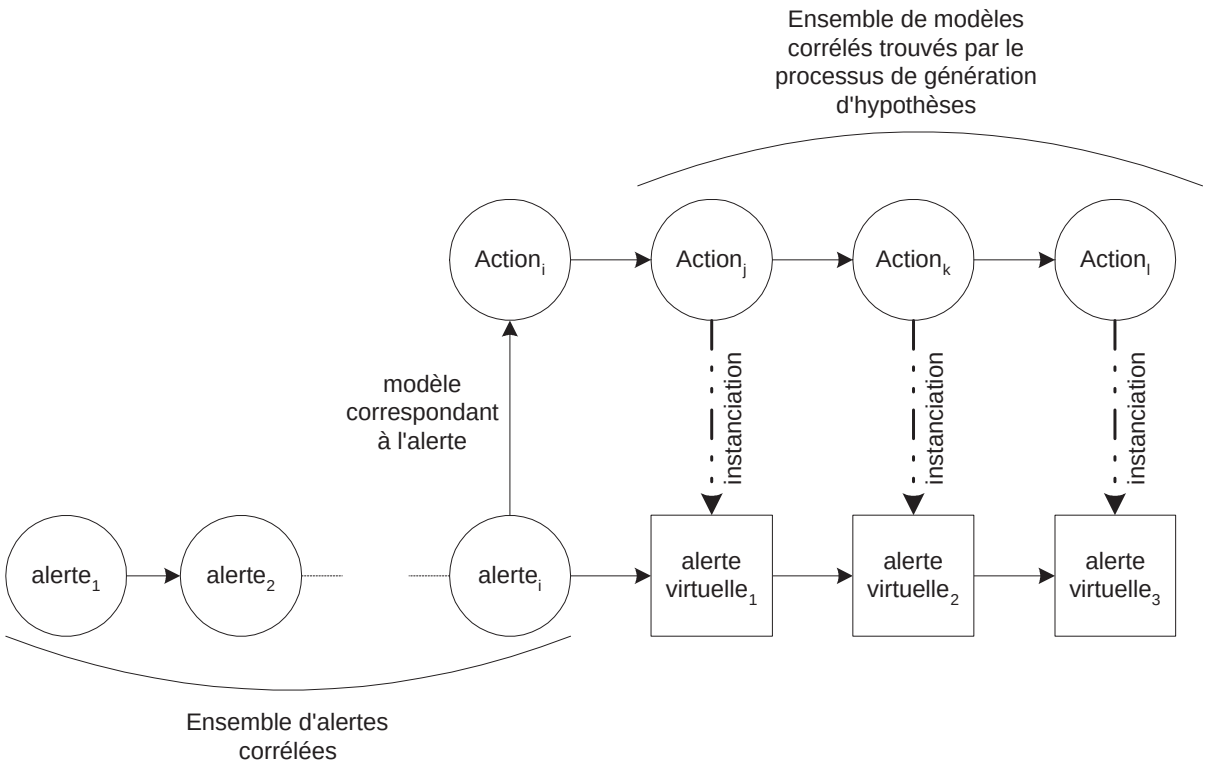

FIG. 4.15 – Principe de la reconnaissance d'intention

4.5 La corrélation pondérée

Nous avons vu que notre approche de la gestion des faux négatifs et le processus de reconnaissance d'intentions génèrent un ensemble de scénarios. Présenter cet ensemble de scénarios à l'administrateur sans donner d'informations supplémentaires sur la plausibilité de chaque scénario ne constitue pas une aide suffisante à la décision. Afin d'être en mesure d'identifier les scénarios les plus plausibles dans cet ensemble, nous proposons d'associer à chaque instance d'action composant un scénario un poids représenté par un réel de l'intervalle $[0, 1]$. Ce poids indique pour une instance d'action A l'influence des autres instances corrélées et à quel point ces instances contribuent à la réalisation de A.

Nous allons tout d'abord exposer comment nous calculons ces poids pour chaque instance d'action puis nous verrons comment agréger les poids d'un ensemble d'actions composant un scénario.

4.5.1 Calcul des poids des instances d'action

Nous définissons le poids d'une instance d'action comme étant le ratio entre le nombre de prédicats satisfaits par d'autres instances d'action et le nombre total de prédicats de la pré-condition.

Dans la suite nous désignons par S_B l'ensemble des actions appartenant au scénario S et qui ont une influence sur B. Alors le poids de corrélation associé à B est défini à partir du nombre de prédicats de $Pre(B)$ qui peuvent être unifiés avec les post conditions des actions de l'ensemble S_B. Plus formellement soit :

- $Pos(S_B) = \bigcup_{A \in S_B} Post(A)$
- $U(S_B, B)$: le nombre de prédicats dans $Pre(B)$ qui sont unifiés avec au moins un élément de $Post(S_B)$

Alors :

Définition 16 *Poids de corrélation*

Le poids de corrélation associé à une action B dans un scénario S, noté par $\omega_S(B)$, est défini de la manière suivante :

$$\omega_S(B) = \begin{cases} 0 & \textit{s'il existe au moins un élément dans } S_B \\ & \textit{ayant une influence négative sur } B \\ 1 & \textit{si } S_B = \emptyset \textit{ (c'est-à-dire } B \textit{ est une action initiale)} \\ \frac{U(S_B,B)}{|Pre(B)|} & \textit{sinon} \end{cases}$$

La notion d'influence négative n'est pas définie dans cette partie de manière formelle. Nous introduisons dans le chapitre 5 suivant la notion d'anti-corrélation pour définir de façon formelle la notion d'influence négative. Intuitivement l'action A a une influence négative sur B, on dit que A anti-corrèle B, si lorsque A est exécutée, B ne peut pas être immédiatement exécutée.

Grâce à la définition 16, nous pouvons introduire la notion de plausibilité d'une instance d'action. Une action ayant un poids de corrélation plus élevé qu'une autre instance d'action lui sera préférée.

4.5.2 Calcul des poids sur les scénarios

Maintenant que nous avons défini comment calculer le poids associé à une instance d'action, nous devons définir comment agréger les poids associés à un scénario composé de plusieurs instances d'action. Une fois le mode d'agrégation défini, nous pouvons comparer des scénarios entre eux et induire une relation d'ordre sur un ensemble de scénarios.

Dans la suite, à chaque scénario $S = (A_1, A_2, ..., A_n, O)$ nous associons son vecteur de poids $(\omega_S(A_1), \omega_S(A_2), ..., \omega_S(A_n), \omega_S(O))$. Les questions qui se posent sont comment agréger ces poids afin d'évaluer la plausibilité d'un scénario donné et comment comparer deux scénarios pondérés. Nous notons g l'opérateur d'agrégation.

Le premier mode d'agrégation le plus naturel est de considérer l'opérateur moyenne, c'est-à-dire :

Mode d'agrégation par la moyenne :

$$g(A_1, ..., A_n, O) = \frac{\sum_{i=1}^{n} \omega_S(A_i) + \omega_S(O)}{n+1}$$

Cependant ce mode d'agrégation est indésirable car des scénarios ayant des poids très différents pour chaque action les composant pourraient être considérés comme étant également plausibles.

Mode d'agrégation conjonctif :

Une condition naturelle que g doit satisfaire est : si $\exists i \in \{1, ..., n\}$ $\omega_S(A_i) = 0$ ou $\omega_S(O) = 0$ alors $g(A_1, ..., A_n, O) = 0$. Les fonctions d'agrégation satisfaisant cette condition sont appelées les opérateurs conjonctifs. Une forme plus faible de tels opérateurs peut être que si $\omega_S(A_i) = 0$ ou $\omega_S(O) = 0$ alors le scénario S devrait être parmi les scénarios les moins plausibles. La forme la plus faible d'un opérateur conjonctif serait de dire qu'un scénario S ne doit pas être parmi les scénarios les plus plausibles si $\omega_S(A_i) = 0$ ou $\omega_S(O) = 0$.

Un exemple d'opérateur d'agrégation conjonctif est l'opérateur minimum :

Définition 17 *Un scénario $S = (A_1, A_2, ..., A_n, O)$ est dit plus plausible que $S' = (B_1, B_2, ..., B_{n'}, O)$ si*

$$min(\omega_S(A_1), \omega_S(A_2), ..., \omega_S(A_n), \omega_S(O)) > min(\omega_{S'}(B_1), \omega_{S'}(B_2), ..., \omega_{S'}(B_{n'}), \omega_{S'}(O))$$

Cette définition considère qu'un scénario est plausible dès lors que le plus petit poids de corrélation d'une action est non nul. Plus le poids est élevé, plus le scénario est plausible.

Cette définition est cependant trop restrictive. Supposons que nous ayons deux scénarios $S = (A_1, A_2, ..., A_n, O)$ et $S' = (B_1, B_2, ..., B_{n'}, O)$. Supposons que le scénario S est tel que $\forall i \in 1, ..., n, \omega_S(A_i) = \alpha$ et $\omega_S(O) = \alpha$. Supposons que le scénario S' est tel que $\exists j$ tel que $\omega_{S'}(B_j) = \alpha$ et que $\forall i \in 1, ..., n', i \neq j, \omega_{S'}(B_i) >$

$\alpha, \omega_{S'}(O) > \alpha$. Dans un tel cas il est clair que l'on préfère le scénario S' au scénario S étant donné que S' contient des actions plus fortement corrélées. Mais, si l'on considère l'opérateur minimum, ces deux scénarios ont la même plausibilité car on ne considère que l'action la moins corrélée.

Un raffinement possible de l'opérateur minimum est d'utiliser l'opérateur dit lexicographe, bien connu dans la théorie du choix social [72]. Cet opérateur n'est défini que lorsqu'il est appliqué à deux vecteurs de même taille. En conséquence, lors de la comparaison de deux scénarios ne comportant pas le même nombre d'actions, nous devons dupliquer le poids le plus faible du scénario le plus court afin d'obtenir deux vecteurs de taille identique. La définition suivante définit l'opérateur "leximin" :

Définition 18 *Soient* $\overrightarrow{v} = (v_1, ..., v_n)$ *et* $\overrightarrow{v}' = (v'_1, ..., v'_n)$ *deux vecteurs de poids ordonnés dans l'ordre croissant des poids, c'est-à-dire* $v_1 > ... > v_n$ *et* $v'_1 > ... > v'_n$. *Alors* $\overrightarrow{v}$ *est dit "leximin" préféré à* $\overrightarrow{v}'$, *noté* $\overrightarrow{v} >_{leximin} \overrightarrow{v}'$, *si* $\exists i$ *tel que* $v_i > v'_i$ *et* $\forall j < i$, $v_j = v'_j$.

Afin d'appliquer cette définition pour ordonner des scénarios, nous représentons les poids de corrélation associés à chaque scénario comme un vecteur de poids $\overrightarrow{v_S}$. Par $\overrightarrow{v_{\sigma(S)}}$ nous désignons le vecteur obtenu à partir de $\overrightarrow{v_S}$ en ordonnant les poids dans l'ordre croissant.

Dès lors, le choix des scénarios plausibles est donné par la définition suivante :

Définition 19 *Un scénario* S *est préféré à* S', *noté* $S > S'$, *si* $\overrightarrow{v_{\sigma(S)}} >_{leximin} \overrightarrow{v_{\sigma(S')}}$. *Un scénario* S *est parmi les scénarios les plus plausibles s'il n'y a pas de scénario* S' *tel que* $S' > S$.

Remarquons que cette définition ne favorise pas systématiquement les scénarios contenant le plus d'actions. En effet, considérons deux scénarios, le premier contenant plus d'actions que le deuxième. Le premier scénario peut cependant être écarté s'il contient par exemple une relation d'anti-correlation.

Nous allons maintenant voir comment nous appliquons la notion de corrélation pondérée dans le processus de corrélation à travers un exemple de scénario.

4.6 Application

Nous prenons l'exemple d'un scénario aboutissant à un accès illégal à un fichier protégé [20, 12].

Considérons un intrus, *bad_guy*, et un fichier confidentiel $secret\_file$. Supposons que *bad_guy* veuille atteindre l'objectif d'intrusion $illegal\_file\_access(secret\_file)$. Le système est au départ dans l'état suivant :

- $file(secret\_file)$
- not$(read_access(bad_guy, secret_file))$
- $printer(ppt)$
- $physical\_access(bad\_guy, ppt)$
- not$(blocked(ppt)$

Ce qui signifie que $secret\_file$ est un fichier, bad_guy n'a pas les droits pour le lire et ppt est une imprimante non bloquée à laquelle bad_guy a un accès physique. bad_guy veut atteindre un état du système dans lequel les prédicats suivants sont vrais :

- $read\_access(bad\_guy, secret\_file)$)
- not($authorized(bad\_guy, read, secret\_file)$)

C'est-à-dire bad_guy peut lire le fichier confidentiel $secret\_file$ alors qu'il n'en a pas le droit. La figure 4.16 montre les modèles d'actions que nous utilisons pour définir ce scénario.

Supposons que les instances d'actions suivantes sont détectées :

- $A = touch(bad\_guy, guy\_file)$ avec $Detectime(A) = t_1$
- $B = block(bad\_guy, ppt)$ avec $Detectime(B) = t_2$
- $C = lpr\text{-}s(bad\_guy, ppt, guy\_file)$ avec $Detectime(C) = t_3$
- $D = remove(bad\_guy, guy\_file)$ avec $Detectime(D) = t_4$
- $E = ln\text{-}s(bad\_guy, guy\_file, secret\_file)$ avec $Detectime(E) = t_5$
- $F = unblock(bad\_guy, ppt)$ avec $Detectime(F) = t_6$
- $G = print\text{-}process(ppt, guy\_file)$ avec $Detectime(G) = t_7$
- $H = get\text{-}file(bad\_guy, secret\_file)$ avec $Detectime(H) = t_8$

Les dates des actions sont telles que $t_1 < t_2 < t_3 < t_4 < t_5 < t_6 < t_7 < t_8$.

4.6.1 Scénarios possibles

A partir de l'ensemble des actions instanciées présentées ci-dessus et à partir de la définition de la corrélation d'actions, nous pouvons construire plusieurs scénarios plausibles qui sont tous corrélés avec l'objectif d'intrusion constitué par l'accès illégal à un fichier. La figure 4.17 montre les liens de corrélation existants dans notre exemple et l'ordre chronologique de leur exécution.

Analyse hors ligne

Nous avons dit précédemment que la corrélation d'actions peut être utilisée dans deux approches différentes. La première approche consiste à considérer l'ensemble des 8 instances d'actions de l'exemple et à générer l'ensemble des scénarios expliquant ces actions. Une fois l'ensemble des scénarios possibles générés, il nous reste à choisir, dans cet ensemble, les scénarios les plus plausibles.

De la figure 4.17 nous pouvons déduire l'ensemble des scénarios possibles à partir des 8 actions observées :

- scénario 1 : $S_1 = (A, B, C, D, E, F, G, H, O)$
- scénario 2 : $S_2 = (A, C, G, H, O)$
- scénario 3 : $S_3 = (A, D, E, G, H, O)$
- scénario 4 : $S_4 = (B, F, G, H, O)$
- scénario 5 : $S_5 = (A, C, D, E, G, H, O)$
- scénario 6 : $S_6 = (A, B, D, E, F, G, H, O)$
- scénario 7 : $S_7 = (A, B, C, F, G, H, O)$

Action $touch(Agent, File)$
Pre : $true$
Post : $file(File)$,
$authorized(Agent, read, File)$,
$authorized(Agent, write, File)$

Action $block(Agent, Printer)$
Pre : $printer(Printer)$,
$physical\_access(Agent, Printer)$,
not($blocked(Printer)$)
Post : $blocked(Printer)$

Action $lpr\ -s(Agent, Printer, File)$
Pre : $printer(Printer)$,
$file(File)$,
$authorized(Agent, read, File)$
Post : $queued(File, Printer)$

Action $remove(Agent, File)$
Pre : $authorized(Agent, write, File)$,
$file(File)$
Post : not ($file(File)$)

Action $ln\ -s(Agent, Link, File)$
Pre : not ($file(Link)$),
$file(File)$
Post : $linked(Link, File)$

Action $unblock(Agent, Printer)$
Pre : $printer(Printer)$,
$blocked(Printer)$,
$physical\_access(Agent, Printer)$
Post : not ($blocked(Printer)$)

Action $print\text{-}process(Printer, Link)$
Pre : $queued(Link, Printer)$,
$linked(Link, File)$,
not ($blocked(Printer)$)
Post : $printed(Printer, File)$,
not ($queued(Link, Printer)$)

Action $get\text{-}file(Agent, File, Printer)$
Pre : $printed(Printer, File)$,
$physical\_access(Agent, Printer)$
Post : $read\_access(Agent, File)$

FIG. 4.16 – Actions utilisées dans le scénario *illegal file access*

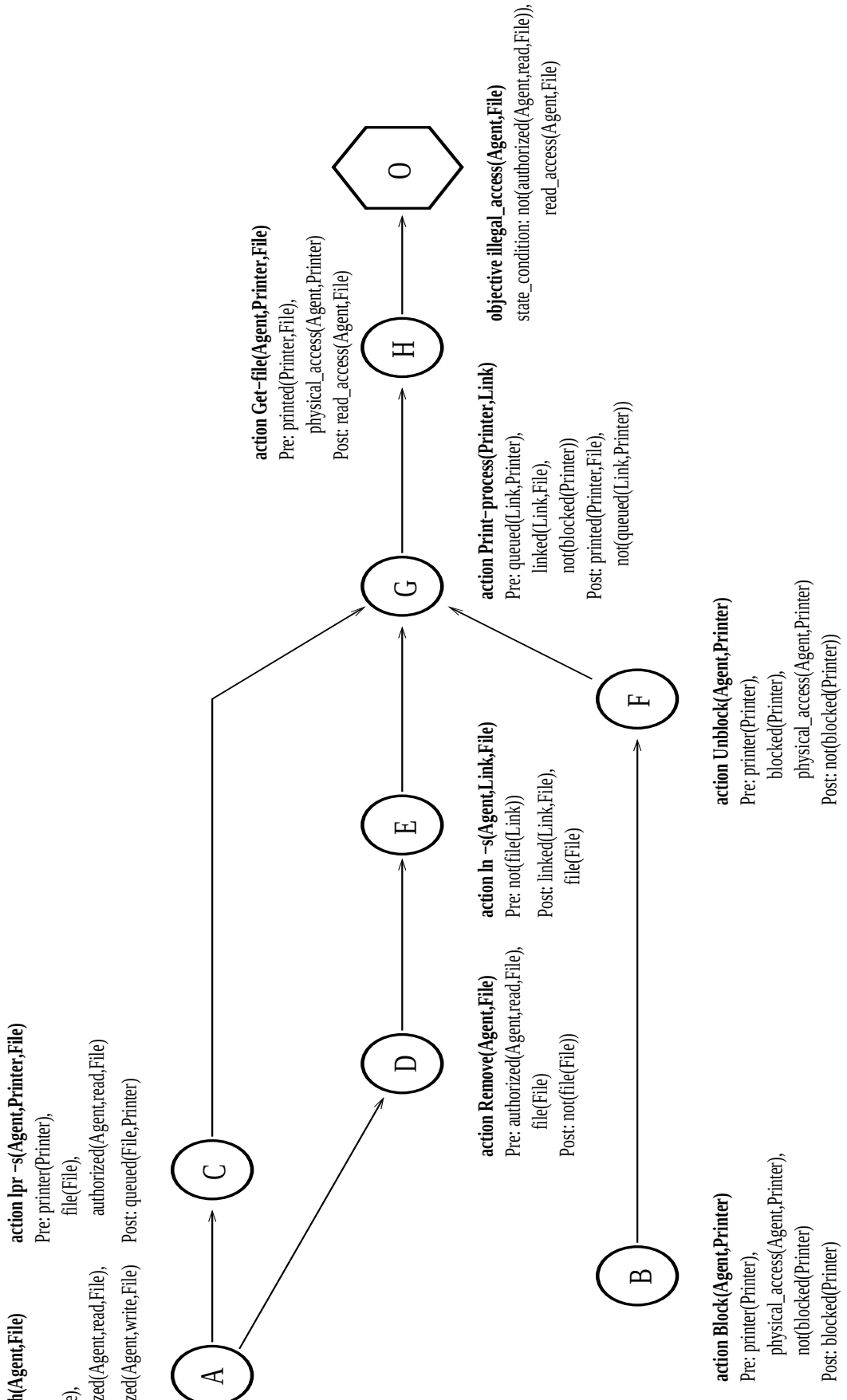

FIG. 4.17 – Graphe de corrélation du scénario d'accès non autorisé à un fichier

Il est à noter que seul le scénario 1 fait intervenir toutes les actions instanciées $(A - H)$. A et B sont des actions initiales car $Pre(A)$ est vraie et tous les prédicats de $Pre(B)$ sont satisfaits par l'état initial du système.

Un administrateur système parviendrait à la conclusion que le scénario le plus dangereux parmi les 7 scénarios possibles est le premier, qui utilise toutes les actions détectées. Cela ne veut pas dire que le scénario maximal faisant intervenir toutes les instances d'action est le plus plausible mais dans notre exemple c'est celui qui a le plus de chances de réussir. En effet, le scénario 1 est le scénario où toutes les pré-conditions des actions sont satisfaites ou corrélées. Par exemple, pour l'action G du scénario 1, toutes les pré-conditions sont corrélées, alors que dans le scénario 4, un seul prédicat est corrélé.

D'après la définition 16, leurs vecteurs de poids correspondants sont :

- scénario 1 : $\overrightarrow{v_{S_1}} = (1, 1, 1, 1, 1, \frac{1}{2}, 1, \frac{1}{2}, \frac{1}{2})$ et $\overrightarrow{v_{\sigma(S_1)}} = (\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, 1, 1, 1, 1, 1, 1)$
- scénario 2 : $\overrightarrow{v_{S_2}} = (1, 1, \frac{1}{3}, \frac{1}{2}, \frac{1}{2})$ et $\overrightarrow{v_{\sigma(S_2)}} = (\frac{1}{3}, \frac{1}{2}, \frac{1}{2}, 1, 1)$
- scénario 3 : $\overrightarrow{v_{S_3}} = (1, 1, 1, \frac{1}{3}, \frac{1}{2}, \frac{1}{2})$ et $\overrightarrow{v_{\sigma(S_3)}} = (\frac{1}{3}, \frac{1}{2}, \frac{1}{2}, 1, 1, 1)$
- scénario 4 : $\overrightarrow{v_{S_4}} = (1, \frac{1}{2}, \frac{1}{3}, \frac{1}{2}, \frac{1}{2})$ et $\overrightarrow{v_{\sigma(S_4)}} = (\frac{1}{3}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, 1)$
- scénario 5 : $\overrightarrow{v_{S_5}} = (1, 1, 1, 1, \frac{1}{3}, \frac{1}{2}, \frac{1}{2})$ et $\overrightarrow{v_{\sigma(S_5)}} = (\frac{1}{3}, \frac{1}{2}, \frac{1}{2}, 1, 1, 1, 1)$
- scénario 6 : $\overrightarrow{v_{S_6}} = (1, 1, 1, 1, \frac{1}{2}, \frac{1}{3}, \frac{1}{2}, \frac{1}{2})$ et $\overrightarrow{v_{\sigma(S_6)}} = (\frac{1}{3}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, 1, 1, 1, 1)$
- scénario 7 : $\overrightarrow{v_{S_7}} = (1, 1, 1, \frac{1}{2}, \frac{1}{3}, \frac{1}{2}, \frac{1}{2})$ et $\overrightarrow{v_{\sigma(S_7)}} = (\frac{1}{3}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, 1, 1, 1)$

D'après la définition 18, les 7 scénarios sont ordonnés de la manière suivante :

$S_1 > S_6 > S_5 > S_7 > S_3 > S_2 > S_4$

Donc le scénario S_1, qui fait intervenir toutes les actions instanciées, est le scénario retenu.

Analyse en ligne

L'approche en ligne diffère peu de l'approche hors ligne excepté le fait que nous ne comparons pas des scénarios nécessairement complets. Par scénario incomplet nous désignons un scénario ne comportant pas encore d'objectif d'intrusion. Un tel scénario est généré par l'algorithme de corrélation qui, à partir des observations (les alertes), génère des suites d'actions compatibles avec les observations. Le nombre maximum d'actions anticipées par l'algorithme est limité pour des raisons de coût de calcul. Par exemple, considérons le cas de notre scénario d'accès illégal à un fichier. Une fois que nous avons reçu la première alerte correspondant à l'action *Touch*, si nous fixons à 1 le nombre limite d'actions anticipées, trois scénarios sont générés. Le premier est composé de *Touch* et *lpr -s*, le deuxième de *Touch* et *Remove* et le troisième de *Touch*, *lpr -s* et *Remove*, *lpr -s* et *Remove* étant tous deux corrélés à *Touch*. Pour dégager le scénario le plus fortement corrélé, il nous suffit d'appliquer les définitions précédentes en omettant dans le calcul du vecteur de poids le poids associé à l'objectif s'il n'est pas présent dans le scénario généré.

4.7 Conclusion

Nous avons présenté comment nous modélisons le processus d'intrusion du point de vue de l'attaquant et comment nous procédons pour détecter l'activités de l'intrus du point de vue de la détection d'intrusions. Nous avons défini la notion de corrélation d'action nous permettant de trouver, dans le flux d'alertes généré par un ensemble de SDI, un sous-ensemble d'alertes correspondant à des actions organisées en scénario. Nous avons défini le mécanisme de génération d'hypothèses qui nous permet de traiter le problème des faux négatifs ainsi que la reconnaissance des intentions de l'attaquant. Cependant la génération d'hypothèses est susceptible de générer un grand nombre de scénarios. Sans information supplémentaire, la génération d'hypothèse n'améliore pas le diagnostic présenté à l'administrateur système. Ainsi, nous avons défini la corrélation pondérée permettant d'induire une relation d'ordre sur un ensemble de scénarios. Cette pondération s'applique aux actions composant un scénario et reflète dans quelle mesure une action est favorisée par l'exécution d'autres actions corrélées.

Chapitre 5

Réaction aux intrusions

La réaction aux intrusions sur un système informatique est une question délicate. En effet, la prise de décision se faisant à partir des alertes de détection d'intrusions, la pertinence du choix des mesures à prendre pour réagir dépend de la qualité des alertes. Ainsi, une alerte peut être un faux positif, auquel cas réagir à une telle alerte n'a pas de sens (sauf si la réaction consiste à éliminer le problème à l'origine du faux positif). D'autre part, même si l'alerte correspond à une action suspecte, (c'est-à-dire qu'elle pourrait être corrélée à une action malveillante), il est possible que cette action ait été exécutée par erreur (par exemple une personne se trompant plusieurs fois de nom de login en essayant de se connecter sur une machine, une entreprise peut scanner accidentellement une machine ne lui appartenant pas, etc...). A la lumière de ces constatations, il parait raisonnable de ne pas adopter par défaut un mécanisme de réaction automatique.

La réaction aux intrusions ne s'envisage que dans le cas du traitement en temps réel des alertes de détection d'intrusions. Le traitement hors-ligne des alertes permet de découvrir des informations sur une éventuelle faiblesse dans la politique de sécurité ou sur l'existence de vulnérabilités. Dans ce cas, une analyse hors-ligne permettra de résoudre éventuellement des problèmes de configuration du système pour que certaines stratégies d'attaque ne soient plus efficaces sur le système. En contrepartie, pendant le temps passé à traiter les alertes hors-ligne et à corriger les problèmes découverts, le système est toujours vulnérable. La fonction de réaction en temps réel trouve ici sa place en permettant de compromettre une intrusion. Dans le cas où un objectif d'intrusion est déjà atteint, la réaction consiste à essayer de restaurer un état du système dans lequel la politique de sécurité est à nouveau respectée.

Ce chapitre est dédié à l'étude des mécanismes de réactions aux intrusions. Nous présentons tout d'abord les travaux existant sur le sujet puis nous présentons notre approche. Nous définissons la notion d'anti-corrélation qui nous permet de sélectionner les contre-mesures adaptées à un scénario donné. Enfin, nous présentons comment nous traitons le cas de la réaction à un scénario inachevé et celui de la réaction à un objectif d'intrusion atteint.

5.1 Travaux existants

Plusieurs travaux de recherche proposent une représentation des contre-mesures disponibles pour un scénario. Dans [69], les auteurs proposent d'exprimer les contre-mesures disponibles pour un scénario dans le champ RESPONSE du langage ADeLe. Cette approche consiste donc à spécifier une liste constante de contre-mesures à exécuter pour chaque scénario spécifié. Cela pose quelques problèmes de maintenance et de spécification. Outre le fait que pour chaque scénario il faille spécifier les contre-mesures à exécuter, quand une nouvelle contre-mesure est spécifiée la mise à jour de la base de scénario est nécessaire. Cette mise à jour consiste à chercher les scénarios pour lesquels la nouvelle contre-mesure s'applique et ensuite à mettre à jour les modèles de ces scénarios.

Mis à part LAMBDA, ADeLe est le seul langage à notre connaissance permettant de spécifier les contre-mesures à mettre en œuvre lors de la détection d'un scénario d'intrusion. Dans [15], Bruschi et Rosti proposent un module noyau linux capable de détecter des attaques spécifiées par des signatures. Le module analyse les appels système pour reconnaître des attaques exécutées depuis la machine surveillée pour les bloquer. Par exemple, le module peut détecter qu'un processus est démarré avec en paramètre un shellcode et décider d'arrêter le processus. Les auteurs précisent que ce type de réponse automatique est agressive dans le sens où elle peut empêcher un utilisateur autorisé, non conscient que sa machine est infectée, de travailler. Dans [62], les auteurs proposent une approche basée sur l'analyse dynamique de code. Ils utilisent un interpréteur pour vérifier le code que doit exécuter le processeur. S'il est considéré comme non malveillant, il est stocké dans un buffer contenant le code vérifié et autorisé. Ainsi, les auteurs proposent de bloquer l'exécution de code malveillant, permettant par la même de réagir aux attaques basées sur l'injection de code malveillant. Dans [10], les auteurs utilisent l'architecture $SHIM$([63]) pour détecter les processus d'une machine hébergeant Linux utilisés pour exécuter une attaque. Ils proposent de surveiller, en plus des processus, les ressources (fichiers, connexions, etc...) associés aux processus. Les entités à surveiller et les contre-mesures disponibles pour chaque entité sont spécifiées par l'utilisateur. Les contre-mesures à prendre vis-à-vis des processus impliqués sont choisies grâce à une fonction évaluant le coût de la réaction et la meilleure contre-mesure pour une attaque est exécutée automatiquement. Les contre-mesures disponibles incluent le redémarrage d'un service, le changement de permissions, le blocage d'une connexion. On peut noter que ces travaux ne concernent que la détection d'attaques s'exécutant sur une seule machine.

Dans [76] les auteurs présentent l'approche définie dans l'architecture EMERALD où les contre-mesures sont associées à chaque RO (Resource Object) qui sont des entités de traitement des alertes. Cependant, les auteurs ne donnent aucune précision sur la nature de ces contre-mesures et surtout ne précisent pas comment elles sont choisies et appliquées pendant le processus de détection.

5.2 La notion d'anti-corrélation

Nous avons vu dans la section précédente que les approches existantes de la réaction à une intrusion consistent à spécifier les contre-mesures disponibles pour chaque scénario décrit. Ce type d'approche pose le problème de la maintenance de la base de signature. Nous proposons une approche de la réaction plus globale dans le sens ou il est possible d'associer automatiquement les contre-mesures aux actions modélisées. Ainsi, lorsque de nouvelles contre-mesures sont définies, les actions sur lesquelles elles peuvent s'appliquer sont déterminées automatiquement. De même, lorsque de nouvelles actions sont modélisées, nous sommes en mesure d'associer automatiquement les contre-mesures existantes aux nouvelles actions.

Le langage LAMBDA (voir section 2.4) décrit une action par sa pré-condition et sa post-condition. La pré-condition représentant les conditions que le système doit satisfaire pour qu'une action puisse être exécutée. Il est possible d'empêcher l'exécution d'une action en faisant en sorte que l'une de ses conditions ne soit pas satisfaite. La notion d'anti-corrélation se base sur cette idée pour définir une fonction de réaction aux intrusions ([25, 19]).

5.2.1 Définitions

La notion d'anti-corrélation, ou encore d'influence négative, se définit entre deux actions. Intuitivement, A anti-corrèle B si lorsque A est exécutée, B ne peut pas être immédiatement exécutée. Plus précisément, A anti-corrèle B s'il existe une expression $expr_1$ dans $Post(A)$ et une expression $expr_2$ dans $Pre(B)$ telles que $expr_1$ et $\texttt{not}(expr_2)$ sont unifiables.

Définition 20 *Anti-correlation*

Deux expressions logiques E et F sont anti-corrélées si une des conditions suivantes est satisfaite :

- $\exists i \in [1,m]$, $\exists j \in [1,n]$ *tels que* $expr_{E_i}$ *et* $\texttt{not}\ (expr_{F_j})$ *sont unifiables via un pgu* Θ.
- $\exists i \in [1,m]$, $\exists j \in [1,n]$ *tels que* $\texttt{not}\ (expr_{E_i})$ *et* $expr_{F_j}$ *sont unifiables via un pgu* Θ.

Définition 21 *Influence négative*

Une action A a une influence négative *sur une action B si $Pre(A)$ et $Post(B)$ sont anti-corrélées d'après la définition 20.*

La figure 5.1 montre un exemple d'actions anti-corrélées. Nous prenons l'exemple de l'exécution d'une action du protocole de communication de l'outil d'attaque Trinoo (voir section 6.2.1 pour un exemple de scénario utilisant cet outil). Sur la figure nous pouvons voir que l'action *kill_slave* a une influence négative sur l'action *command_dos*. En effet l'action *kill_slave* permet de fermer le programme permettant de transformer un ordinateur en zombie pour ensuite être utilisé dans une

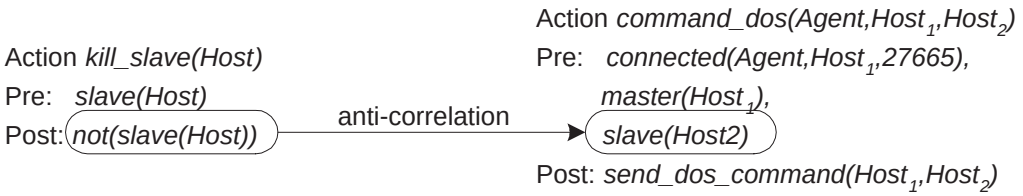

FIG. 5.1 – Exemple de relation d'anti-corrélation entre deux modèles d'actions

attaque distribuée. Ce programme permet d'envoyer un message à un ordinateur maître pour lui signaler que la machine zombie peut être utilisée pour lancer une attaque. Lorsque la machine mitré reçoit le message, elle mémorise que cette machine est une machine esclave. L'action *kill_slave* exploite une commande que reconnaît le programme zombie. Précisément, elle demande au processus de se terminer.

Les contre-mesures ne sont pas modélisées de manière différente des actions. En effet, c'est l'anti-corrélation liant deux actions A et B qui définit le fait que A soit une contre-mesure pour B. Tout comme pour la corrélation, des règles d'anti-corrélation sont générées à partir des modèles des actions. Ces règles sont ensuite utilisées par le moteur de corrélation pour déterminer en temps réel les contre-mesures disponibles pour chaque hypothèse représentée par une alerte virtuelle.

5.2.2 Application de l'anti-corrélation

Dans le cadre d'un processus de réaction en temps réel, nous pouvons distinguer deux cas :

- Un scénario d'intrusion a été complètement observé et l'objectif d'intrusion associé est atteint : dans ce cas nous voulons ramener le système dans un état où la politique de sécurité est à nouveau respectée. Au niveau de notre modélisation des objectifs d'intrusion, faire en sorte que la politique de sécurité soit à nouveau respectée revient à invalider la condition sur l'état du système associé à l'objectif.
- Un scénario d'intrusion est en cours de réalisation : à partir des observations correspondant à cette partie de scénario, un ensemble d'actions possibles débouchant sur un objectif d'intrusion peut avoir été identifié. Afin de contrecarrer la progression du scénario, il nous faut empêcher l'exécution de l'une de ses actions.

Réponse à un objectif d'intrusion

La réponse à la réalisation d'un objectif d'intrusion consiste à mettre à jour l'état du système pour invalider la condition associée à l'objectif. La condition sur l'état du système d'un objectif d'intrusion est une conjonction de prédicats de la forme $Cond_1 \wedge Cond_2 \wedge ... \wedge Cond_n$ (définition 8 de la section 4.2.3). L'invalidation de la

condition d'un objectif consiste à invalider au moins un des prédicats $Cond_i$.

Prenons l'exemple d'un objectif d'intrusion O et de deux actions A et B corrélées à O. Supposons que l'exécution de A et B a permi d'atteindre O. Supposons maintenant que deux contre-mesures R_0 et R_1 existent pour l'objectif O (figure 5.2). Enfin, supposons que les actions A et B sont instanciées avec les paramètres $X = x$ et $Y = y$. Pour appliquer ces contre-mesures et invalider la condition de O, il faut instancier les variables de R_0 et R_1. Cela peut être fait en utilisant les *pgu* Ξ_{AO} et Ξ_{BO} représentant respectivement les règles de corrélation entre A, B, et O et les *pgu* Ψ_{R_0O} et Ψ_{R_1O} représentant les règles d'anti-corrélation entre R_0, R_1 et O. En effet, de Ξ_{AO} et Ψ_{R_0O} nous pouvons déduire que $X = X' = X'' = x$. De même de Ξ_{BO} et Ψ_{R_1O} nous déduisons que $Y = Y' = Y'' = y$.

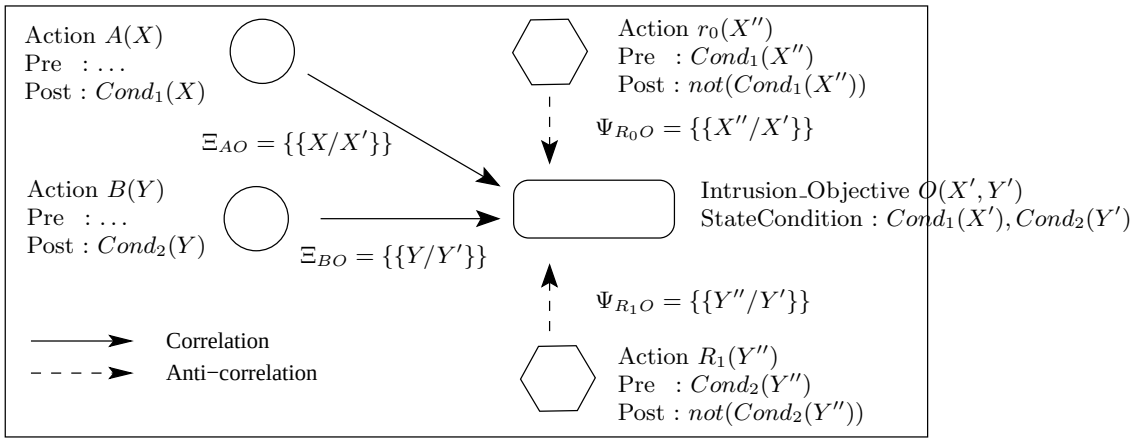

FIG. 5.2 – Graphe de corrélation avec réponse directe sur un objectif

Cet exemple présente des contre-mesure n'étant anti-corrélé avec l'objectif que par un seul prédicat. Dans le cas général ou un ensenble de contre-mesures anti-corrèle un objectif par un nombre arbitraire de prédicats, nous présentons dans la section 5.3 une méthode permettant de choisir la contre-mesure la plus adaptée.

Réponse à un scénario inachevé

Nous avons vu que les contre-mesures peuvent être utilisées pour tenter d'annuler les effets d'un scénario d'attaque ayant atteint un objectif d'intrusion. L'opération consistant à modifier l'état du système afin de revenir dans un état où la politique de sécurité est respectée n'étant pas chose facile, il est préférable d'empêcher un attaquant d'atteindre son but une fois le début de son intrusion détectée. La figure 5.3 montre un exemple de scénario partiellement observé et un ensemble d'hypothèses générées aboutissant à un objectif d'intrusion. Les rectangles représentent les contre-mesures trouvées pour chaque hypothèse (voir section 4.4 pour plus de détails sur la génération d'hypothèses). Nous n'avons représenté qu'une seule contre-mesure par hypothèse mais il se peut qu'il en existe plusieurs. Le principe de la réaction à un scénario en cours d'exécution est de proposer à l'administrateur un ensemble de

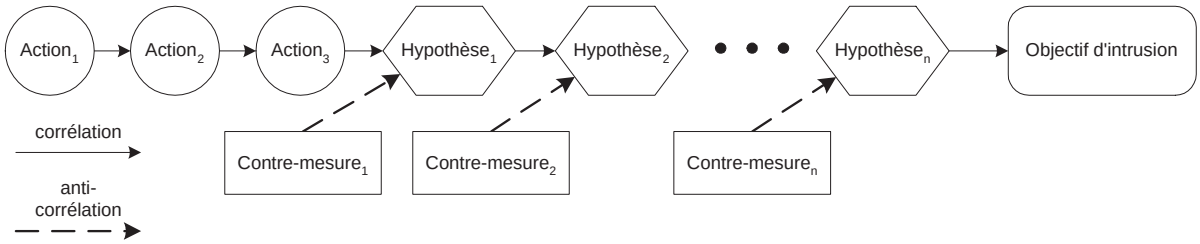

FIG. 5.3 – Scénario en cours d'exécution et les hypothèses

contre-mesures permettant d'empêcher la réalisation des hypothèses. Nous laissons à l'administrateur le soin de choisir ou non d'activer les contre-mesures. Cependant, notre implantation peut lancer automatiquement une contre-mesure si l'administrateur l'a spécifié.

Notre exemple ne concerne que le cas où il n'existe qu'une seule contre-mesure par hypothèse. Dans le cas général il est possible d'avoir n contre-mesures pour une hypothèse. Dans ce cas, il est nécessaire d'aider l'administrateur à choisir la contre-mesure la mieux adaptée. Par mieux adaptée nous faisons référence à la contre-mesure la plus susceptible de stopper l'avancement du scénario. Nous proposons un critère de sélection dans la section suivante.

Il est à noter que le champ *Detection* d'un modèle d'action utilisé comme contre-mesure n'est pas évalué lors de l'instanciation de l'action. En effet, nous avons vu dans les sections précédentes que les *pgu* sont utilisés pour déduire d'un objectif ou d'une alerte virtuelle les paramètres d'une action utilisée comme contre-mesure. Dans le chapitre 6, nous ometons systèmatiquement de préciser le champ *Detection* lorsque nous donnons les modèles d'actions utilisées comme contre-mesures.

5.3 Choix de la contre-mesure

Afin de choisir la contre-mesure pour une action probable faisant partie d'un scénario aboutissant sur un objectif d'intrusion, nous devons définir un critère de sélection. Intuitivement, si nous avons à choisir dans une liste de contre-mesures, nous sélectionnons celle qui a le plus de chance d'empêcher l'exécution de l'action correspondant à l'hypothèse. Cela se traduit au niveau de la corrélation par une contre-mesure anti-corrélant le plus de prédicats possible dans la pré-condition de l'action à empêcher.

Pour ce faire nous proposons la notion d'anti-corrélation pondérée. Le principe se rapproche de la corrélation pondérée mais ne s'applique pas sur une instance d'action mais sur une hypothèse. Nous proposons de mesurer dans quelle mesure une contre-mesure C a une influence négative sur une action A en comptant le nombre de prédicats de $pre(A)$ anti-corrélés par C.

Soit $A_c(A_1, A_2)$ le nombre de prédicats de $post(A_1)$ anti-corrélés avec $pre(A_2)$.

Définition 22 *Poids d'anti-corrélation*

le poids d'anti-corrélation associé à une contre-mesure C pour une hypothèse H, noté par $\chi_H(C)$, est défini de la manière suivante :

$$\chi_H(C) = \begin{cases} 0 & \text{si } post(C) \text{ n'est pas anti-corrélé avec } pre(H) \\ 1 & \text{si tous les prédicats de } pre(H) \text{ sont anti-corrélés par } post(C) \\ \frac{A_c(C,H)}{|Pre(H)|} & \text{sinon} \end{cases}$$

Ainsi nous pouvons ordonner une liste de contre-mesures pour une hypothèse H par ordre décroissant d'efficacité en fonction du poids d'anti-corrélation.

5.4 Conclusion

Nous avons présenté dans ce chapitre une approche pour la réaction aux intrusions basée sur la notion d'anti-corrélation. Grâce au mécanisme défini, il est possible de proposer à l'administrateur système un ensemble de contre-mesures permettant de faire face à deux situations :

- la restauration de l'état du système après une attaque réussie afin de respecter à nouveau la politique de sécurité.
- l'arrêt d'une intrusion en cours d'exécution via le lancement de contre-mesures.

Nous adoptons une approche prudente en laissant le choix à l'administrateur de lancer les contre-mesures. Cependant, il est possible d'automatiser la tâche du lancement de certaines contre-mesures ne risquant pas de compromettre la disponibilité du système. Notre implantation permet de spécifier les contre-mesures pouvant être lancées automatiquement. La section suivante présente quelques exemples expérimentaux d'application de la notion d'anti-corrélation dans notre implantation.

Chapitre 6

Expérimentation

Ce chapitre présente les expérimentations que nous avons réalisées sur les implantations des sous-modules d'agrégation/fusion et de corrélation. Ces deux modules ont été implantés en C++ et peuvent fonctionner de manière autonome ou intégrés dans la chaîne de traitement du flux d'alertes, comme présenté dans la section 2.3 du chapitre 2. L'ensemble des fonctions présentées dans la section 2.3 a été implémenté et constitue le module CRIM (Corrélation et Reconnaissance d'Intentions Malveillantes). Nous présentons tout d'abord les résultats obtenus avec l'implantation du sous-module d'agrégation et de fusion d'alertes présenté dans le chapitre 3 puis nous présentons les scénarios modélisés et détectés grâce au sous-module de corrélation présentée dans le chapitre 4.

6.1 Module d'agrégation et de fusion

Le module d'agrégation et de fusion a été implanté dans deux versions. La première a été réalisée par F.Cuppens et implante l'agrégation et la fusion présentées dans [18]. Elle est intégrée dans le même module que la première version du moteur de corrélation que nous présentons dans la section 6.2. Cette version nécessite un apprentissage pour pouvoir regrouper des alertes qui ont été générées lors de la détection du même événement. Le langage utilisé est Prolog.

J'ai réalisé la deuxième implantation du module d'agrégation et de fusion en utilisant le langage C++. Le module lit des alertes IDMEF et génère une alerte IDMEF par cluster d'alertes similaires. L'expérimentation que nous avons effectuée a consisté à jouer des attaques sur une machine équipée d'une sonde *Snort* générant des alertes IDMEF. Nous avons ensuite étudié les alertes générées par *Snort* et le résultat de l'agrégation des alertes. Nous avons utilisé Snort dans sa configuration standard, c'est-à-dire en appliquant les règles présentes par défaut dans le fichier de configuration. La seule modification par rapport au Snort standard est l'utilisation du plugin IDMEF pour obtenir les alertes dans ce format.

En ce qui concerne la configuration de l'outil d'agrégation et de fusion, nous n'avons pas défini d'événements dans l'ensemble $\mathcal{E}_{vt}$ (voir section 3.2.1). Cela veut dire que les poids utilisés lors de la comparaison de deux alertes sont toujours les

mêmes, quel que soit l'événement à l'origine de l'alerte, et sont tous fixé à 1 par défaut. D'autre part, nous avons arbitrairement fixé le delai d_1 à deux secondes et d_2 à six secondes. Ainsi nous étudions le comportement de l'outil sans le paramétrer.

Les tests que nous avons réalisés sur le module d'agrégation et de fusion ont été faits à partir de l'outil Nmap. Nmap [44] est un outil gratuit qui permet d'explorer des réseaux. Il permet de balayer rapidement un grand nombre d'adresses IP et de récupérer diverses informations. Ainsi, il est possible, entre autres, de déterminer le système d'exploitation d'une machine, de déterminer les noms et les versions des services démarrés sur cette machine et les firewalls utilisés. Cet outil peut permettre à un administrateur de vérifier les caractéristiques de certaines machines dans son réseau mais peut aussi servir d'outil d'attaque. En effet, il peut permettre à un attaquant de réaliser les premières étapes d'un scénario d'attaque, à savoir la récupération d'informations sur une machine cible ou la recherche d'une cible potentielle. Nous avons décidé d'exécuter quelques commandes Nmap à partir d'une machine vers deux machines cibles. Nous avons utilisé deux machines cibles pour pouvoir exécuter la même commande Nmap au même moment sur les deux cibles. Ainsi, nous vérifions si le module d'agrégation et de fusion est capable de créer des clusters, à partir du flux d'alerte généré par Snort, relatifs aux deux événements se déroulant en parallèle.

Voici les commandes que nous avons utilisées :

– *Nmap -sO* : cette commande permet d'identifier les protocoles utilisés par une machine. Nmap envoie des paquets IP sans en-tête de protocole vers la machine cible sur tous les protocoles existants. Lors de l'exécution de cette commande, snort génère 3776 alertes. L'exécution de la commande prend 120 secondes. Nous avons appliqué l'algorithme d'agrégation en-ligne sur ces alertes et obtenu 64 clusters. Après analyse, nous avons constaté que 32 clusters concernent la première machine cible et les 32 autres la seconde cible. Les 32 clusters ont la même composition pour chaque cible (à quelques alertes près pour les plus gros clusters), ce qui indique que l'outil d'agrégation a pu séparer les alertes relatives aux deux exécutions de Nmap en fonction de la machine visée. Nous étudions ici la composition des 32 clusters concernant la première machine visée.
  Parmis les 32 clusters, nous avons une dizaine de clusters dont la taille varie entre 30 et 400 alertes. A l'intérieur d'un même cluster les alertes sont proches les unes des autres dans le temps, l'écart maximum constaté étant de 10 secondes. Pour ces clusters, la classification des alertes est la même et correspond à l'événement *BAD-TRAFFIC Unassigned/Reserved IP protocol*. Ces alertes correspondent à l'envoi de paquets IP dont le numéro de protocole n'est pas valide. Ce numéro étant codé sur 8 bits, Nmap génère des paquets IP en utilisant les 256 valeurs possibles du numéro de protocole.
  Les autres clusters ont une taille plus faible, entre 1 et 6 alertes. Les alertes ont la même classification à l'intérieur de chacun de ces petits clusters. L'écart temporel maximum au sein de ces petits clusters n'éxcède pas 2 secondes. Pour

chacun de ces clusters, l'événement associé aux alertes correspond à l'envoi de paquets IP dont le numéro de protocole est réservé. Ainsi les classifications retrouvées sont *BAD-TRAFFIC IP Proto 55 IP Mobility*, *BAD-TRAFFIC IP Proto 103 PIM*, *BAD-TRAFFIC IP Proto 77 Sun ND* et *BAD-TRAFFIC IP Proto 53 SWIPE*.
Nous avons ainsi pu isoler des événements ponctuels noyés dans un ensemble d'alertes correspondant à l'utilisation d'un protocole non valide. Après fusion, tous les clusters donnent une simple alerte. Nous constatons que d'un total de 3776 alertes occupant 3000Ko, nous passons à un total de 64 alertes occupant 104Ko. Ce résultat est intéressant dans la mesure où nous avons réduit la quantité d'information à traiter tout en isolant des événements particuliers. Cependant, la taille variable des gros clusters et l'écart temporel important entre la plus récente et la plus vieille alerte au sein d'un de ces gros clusters soulèvent quelques questions. Tout d'abord la façon dont nous calculons la valeur de similarité entre une nouvelle alerte et un cluster n'est peut-être pas la plus pertinente. En effet, nous faisons la moyenne des valeurs de similarité calculées entre la nouvelle alerte et chaque alerte du cluster. Ceci diminue l'impact d'un écart temporel important entre la nouvelle alerte et la plus vieille alerte du cluster. Nous sommes actuellement en train de modifier l'implantation pour qu'à chaque fois qu'une alerte est ajoutée à un cluster, ses informations soient fusionnées avec le cluster. De cette façon, un cluster n'est plus un ensemble d'alertes mais une alerte de fusion. La date de l'alerte de fusion est celle de la première alerte agrégée. D'autre part, cette méthode diminue les calculs nécessaires, la fusion des informations ne prenant pas beaucoup de temps par rapport à la comparaison d'une alerte avec un grand nombre d'alerte.

– *Nmap -sS* : cette commande permet d'effectuer un balayage *TCP-SYN*. Cette action consiste à envoyer des paquets SYN sur un port pour savoir s'il est ouvert. Si le port est ouvert, un paquet *SYN-ACK* est reçu, sinon un paquet RST est reçu. L'exécution de cette commande génère trois alertes. L'écart entre la première et la dernière alerte est d'une seconde. Les trois alertes sont regroupées dans un même cluster étant donné que l'écart temporel est faible, les adresses IP source et cible des trois alertes sont les même et les numéros de port source sont identiques (55290). De plus les numéros de port destination font partie des ports « well known » (voir figure 3.2, section 3.1.2). Après fusion, l'alerte obtenue possède une adresse source et une adresse cible, un numéro de port source et une liste de trois numéros de port cible.

Ces deux exemples montrent que le module d'agrégation et de fusion se comporte de manière satisfaisante dans les deux tests effectués. Nous somme en train de modifier le module pour pouvoir l'interfacer avec une base de données. Ainsi, il est plus facile pour d'autres SDI d'exploiter les alertes de fusion produites par notre module. Le module de corrélation présenté dans la section suivante a fait l'objet de tests plus avancés.

6.2 Module de corrélation

Le module de corrélation a fait l'objet de deux implantations. La première, réalisée par F.Cuppens et A.Miège, est écrite en prolog et réunie les deux modules de fusion et de corrélation. Ce premier prototype n'implante pas la notion de corrélation pondérée ni la notion d'anti-corrélation. Cependant, la génération d'hypothèse est présente. Une interface réalisée en Java permet de visualiser les résultats du traitement des alertes (voir figure 6.1). Ce prototype a été réalisé dans le cadre du projet MIRADOR dont un des buts était l'implantation d'une plateforme multi-SDI. Étant donné le langage utilisé (prolog), les performances de la première implantation ne sont pas très élevées. La vitesse de traitement des alertes étant de l'ordre d'une alerte à la seconde. Dans ce prototype, la base de données représentant K (voir section 4.2.4) est simulée par une base de faits prolog.

J'ai réalisé une seconde implantation du module de corrélation en C++ dans le but d'implanter les mécanismes manquant dans le précédent prototype et pour augmenter les performances. Ainsi ce prototype est environ 50 fois plus rapide que le précédent. Cependant, l'interface avec la base de données étant en cours d'implantation et étant simulée, il est difficile d'estimer l'impact de l'interfaçage avec une vraie base de données. Le prototype lit les alertes au format IDMEF et génère des alertes de scénario et de réaction dans le même format. Ce prototype implante la corrélation pondérée et le mécanisme de réaction se basant sur l'anti-corrélation. Le module de fusion n'est cette fois plus intégré et fonctionne à part, tel qu'il a été présenté dans la section précédente. La figure 6.2 présente l'interface développée pour ce nouveau prototype. Cette figure montre l'exemple de la détection du début d'un scénario d'intrusion du type déni de service distribué.

Nous allons maintenant présenter les scénarios d'intrusion que nous avons modélisés et expérimentés sur ce prototype.

6.2.1 Exemples de scénarios d'intrusion

Cette section présente trois scénarios d'intrusion. Le premier scénario consiste à obtenir un accès illégal à une machine avec les droits d'administrateur. Le deuxième exemple présente un scénario où l'attaquant utilise une attaque de type syn-flooding pour pouvoir créer une adresse forgée afin d'obtenir un accès illégal à une machine avec les droits d'administrateur. Le troisième exemple est une attaque en déni de service distribuée. Pour chaque scénario nous présentons la modélisation des actions, des objectifs d'intrusion et des contre-mesures.

Obtention illégale d'un accès root

Ce premier exemple de scénario [23] fait intervenir cinq actions. Les modèles de ces actions sont représentés sur la figure 6.3. Le principe de cette attaque est d'exploiter une politique de sécurité mal configurée. Plus précisément, cette attaque consiste à exploiter le fait que la partition correspondant au répertoire de l'utilisateur root d'une machine puisse être exportée. Les cinq étapes de l'attaque sont les

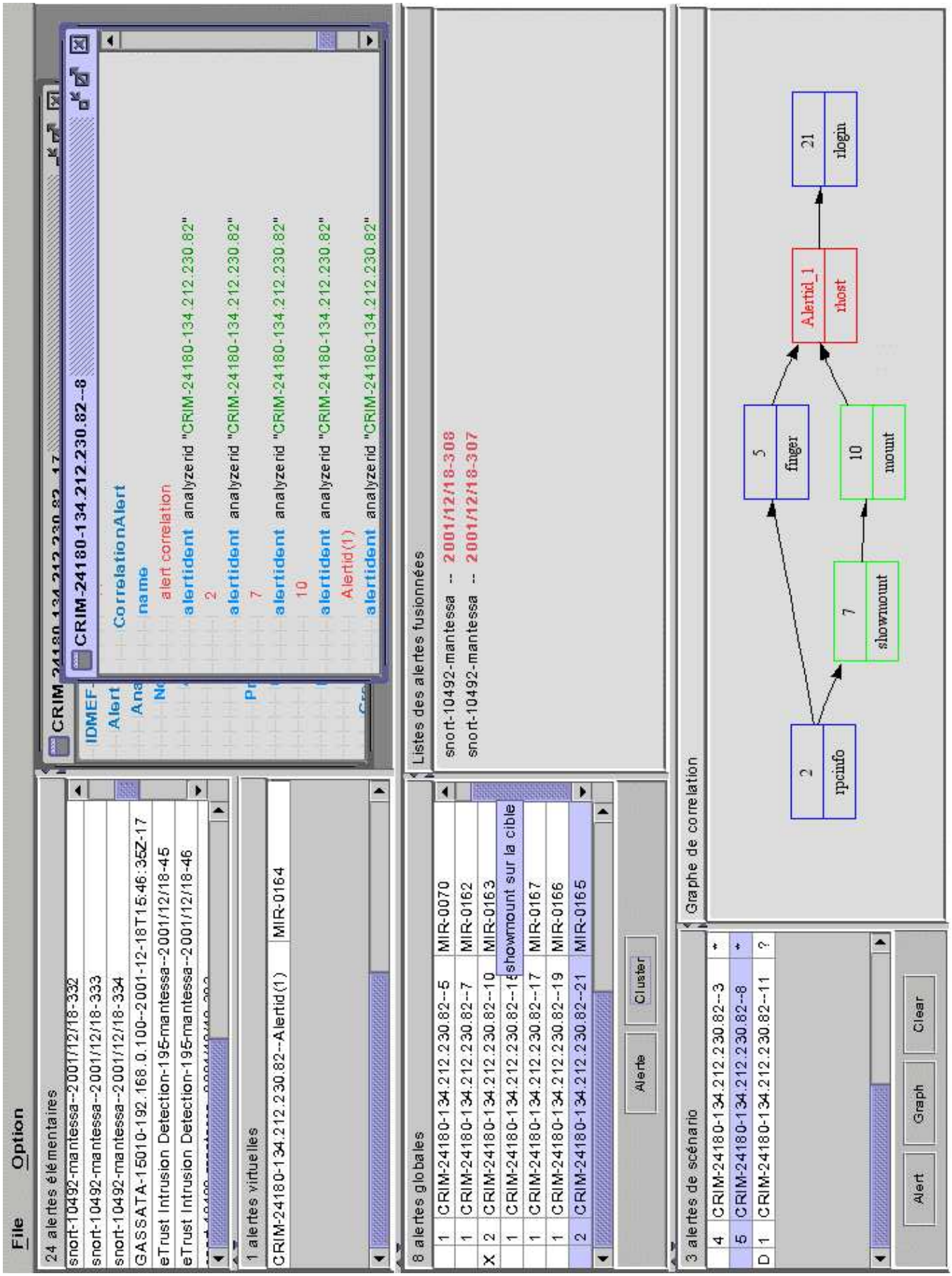

FIG. 6.1 – Interface Java du premier prototype réunissant les deux modules de fusion et de corrélation.

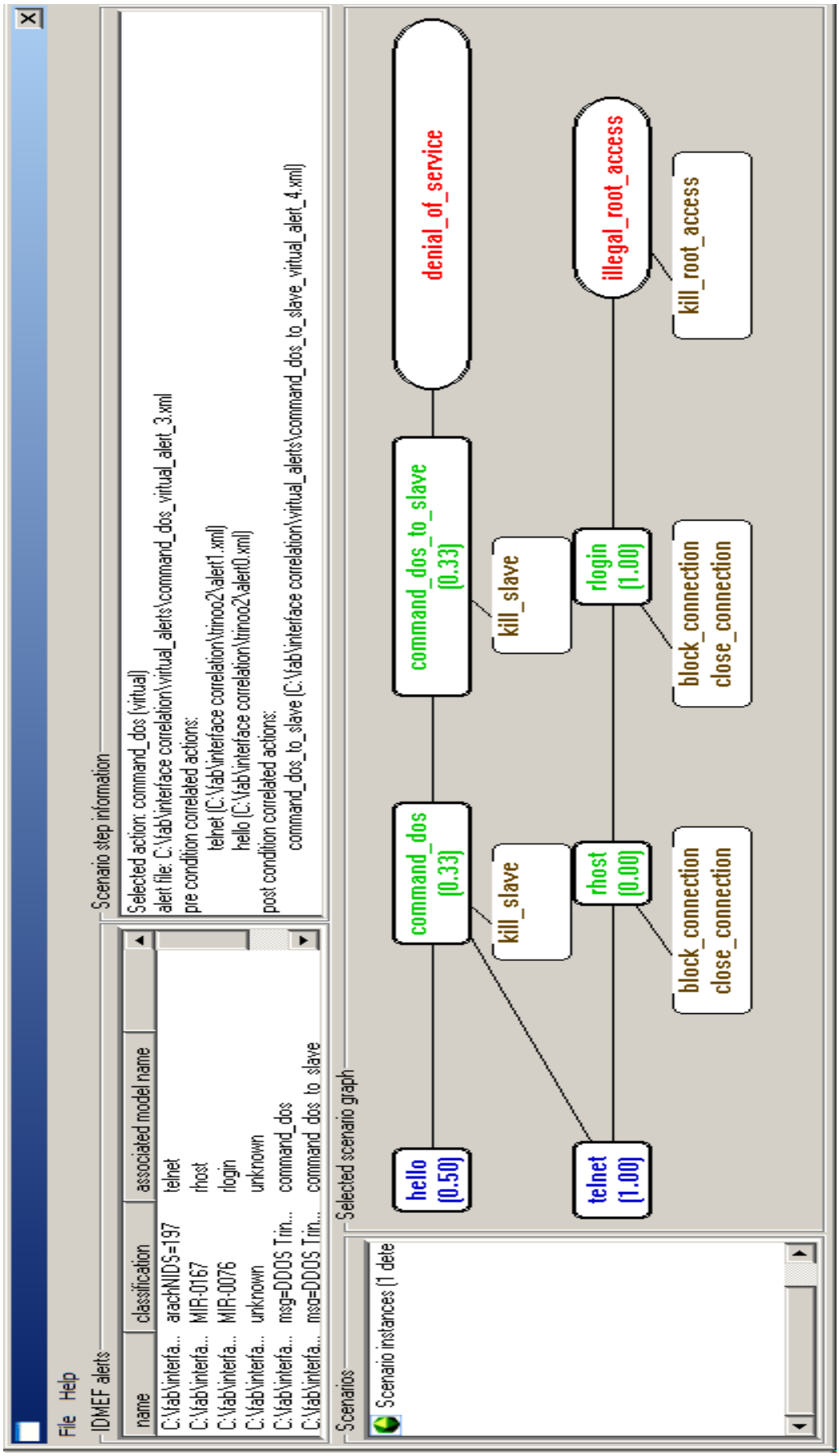

FIG. 6.2 – Interface du module de corrélation écrit en C++.

suivantes :

- Action 1 : *rpcinfo sur la cible*
  L'attaquant exécute la commande *rpcinfo* sur une machine pour savoir si le service mountd permettant d'exporter des partitions est démarré sur la cible.

- Action 2 : *showmount sur la cible*
  L'attaquant exécute la commande *showmount* sur la machine cible pour connaître les partitions exportables.

- Action 3 : *montage d'une partition exportable*
  L'attaquant exécute la commande *mount* pour monter une partition exportable correspondant au répertoire de l'utilisateur root de la machine cible.

- Action 4 (non détectable) : *modification du fichier .rhost de la cible*
  L'attaquant modifie le fichier *.rhost* de la machine cible sur la partition qu'il vient de monter sur son système. Cette action n'est pas détectable car exécutée localement sur la machine de l'attaquant, elle constitue donc un faux négatif. L'attaquant ajoute la chaîne de caractères "++" au fichier .rhost pour pouvoir se connecter à la cible sans authentification.

- Action 5 : *rlogin sur la cible*
  L'attaquant exécute la commande *rlogin* afin d'obtenir un accès à la cible avec les privilèges de l'utilisateur root. Il n'a pas besoin de s'authentifier grâce aux effets de l'action précédente.

La dernière étape permet d'atteindre l'objectif d'intrusion *illegal_root_access* présenté dans le chapitre 4, section 4.2.3. Grâce à la génération d'hypothèse, il est possible de détecter le scénario entier malgré le fait que l'action correspondant à la modification du fichier *.rhost* ne soit pas détectée. Une alerte virtuelle correspondant à cette action est créée et représente l'hypothèse générée. La figure 6.5 montre le résultat de la détection de ce scénario par notre implantation. Les actions sont représentées dans l'ordre dans lequel elles arrivent, de gauche à droite. Les actions sont affichées en bleu et les hypothèses en vert. Les poids de corrélation sont affichés en dessous du nom des actions. À droite est représenté l'objectif d'intrusion en gris foncé. Sur cette image, le fait que l'objectif d'intrusion soit affiché en gris foncé signifie qu'il a été atteint. Les contre-mesures disponibles pour cet objectif, déterminées par le moteur de corrélation grâce aux règles d'anti-corrélation, sont affichées sous l'objectif d'intrusion en marron. Pour l'objectif d'intrusion *illegal_root_access* , le moteur de corrélation trouve une contre-mesure. Cette contre-mesure permet d'annuler un login à distance en tant qu'utilisateur root. Le module permettant de réaliser cette contre-mesure est présenté dans [22]. Le modèle de cette contre-mesure est représenté sur la figure 6.4.

Action $rpcinfo(User, Address)$
Pre : $remote\_access(User, Address)$,
$use\_service(Address, mountd)$
Post : $knows(User, use\_service(Address, mountd))$
Detection : *classification(Alert,'rpcinfo')*
$source(Alert, User)$
$target(Alert, Address)$

Action $showmount(User, Address)$
Pre : $remote\_access(User, Address)$,
$use\_service(Address, mountd)$,
$mounted\_partition(Address, Partition)$
Post : $knows(User, mounted\_partition(Address, Partition))$
Detection : *classification(Alert,'showmount')*
$source(Alert, User)$
$target(Alert, Address)$

Action $mount(User, Address, Partition)$
Pre : $remote\_access(User, Address)$,
$mounted\_partition(Address, Partition)$
Post : $can\_access(User, Partition)$
Detection : *classification(Alert,'mount')*
$source(Alert, User)$
$target(Alert, Address)$
$target\_partition(Alert, Partition)$

Action $rhost(User, Address, Partition)$
Pre : $remote\_access(User, Address)$,
$can\_access(User, Partition)$,
Post : $user\_access(User, Address)$
Detection : *classification(Alert,'.rhost modification')*
$source(Alert, User)$
$target(Alert, Address)$
$target\_partition(Alert, Partition)$

Action $rlogin(User, Address)$
Pre : $remote\_access(User, Address)$,
$user\_access(User, Address)$
Post : $root\_access(User, Address)$
Detection : *classification(Alert,'rlogin')*
$source(Alert, User)$
$target(Alert, Address)$

FIG. 6.3 – Modèles des actions du scénario d'accès illégal à une machine avec les droits d'administrateur

Action $block\_connection(User, Address)$ Pre : $remote\_access(User, Address)$ Post : $not(remote\_access(User, Address))$
Action $close\_connection(User, Address)$ Pre : $remote\_access(User, Address)$ Post : $not(remote\_access(User, Address))$
Action $kill\_root\_access(Agent, Address)$ Pre : $remote\_access(User, Address)$ $root\_access(Agent, Address)$ Post : $not(root\_access(Agent, Address))$

FIG. 6.4 – Modèles des contre-mesures du scénario d'accès illégal à une machine avec les droits d'administrateur

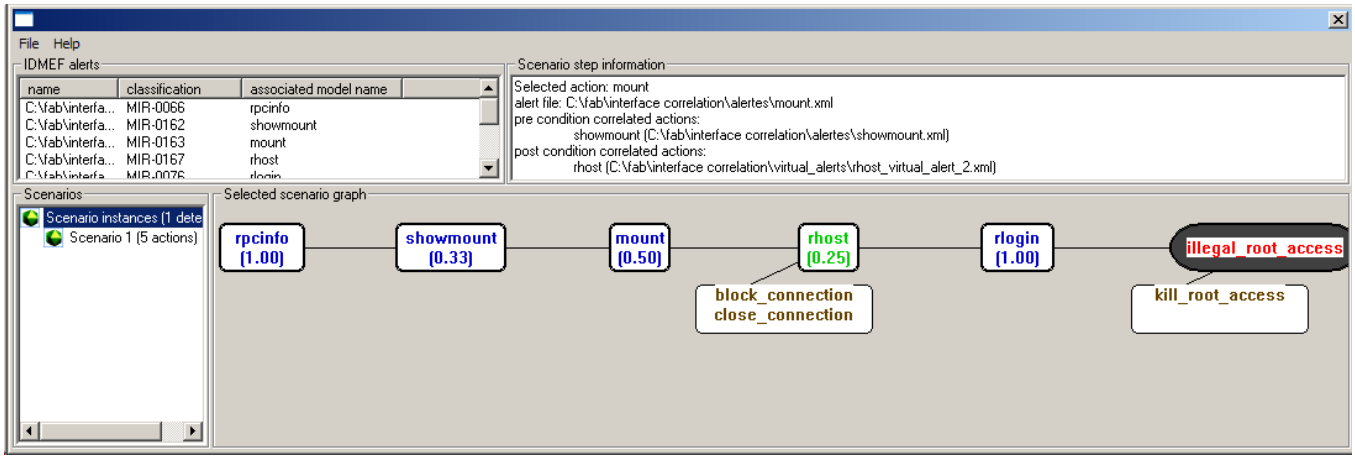

FIG. 6.5 – Détection du scénario d'accès illégal à une machine avec les droits d'administrateur.

La figure 6.5 montre la détection du scénario complet. Dans le chapitre 5, nous avons présenté comment nous réagissons quand un objectif d'intrusion est atteint. Nous allons maintenant voir comment notre implantation détecte le scénario au fur et à mesure qu'il se déroule. La figure 6.6 montre l'interface de notre implantation lorsque l'alerte correspondant à la première action de ce scénario ($rpcinfo$) est reçue.

La première action du graphe est affichée en bleue pour indiquer qu'elle correspond à la première alerte reçue. Les autres actions sont représentées en vert car elles correspondent aux hypothèses générées par le moteur de corrélation. Un objectif d'intrusion corrélé à la dernière hypothèse générée est identifié et affiché en blanc pour indiquer qu'il n'est pas encore atteint. Pour chaque hypothèse générée, le moteur de corrélation détermine l'ensemble des contre-mesures disponibles. Pour le scénario présenté, deux contre-mesures sont disponibles pour chaque hypothèse. Les modèles de ces contre-mesures sont représentées sur la figure 6.4. Le graphe de corrélation de ce scénario est représenté sur la figure 6.7.

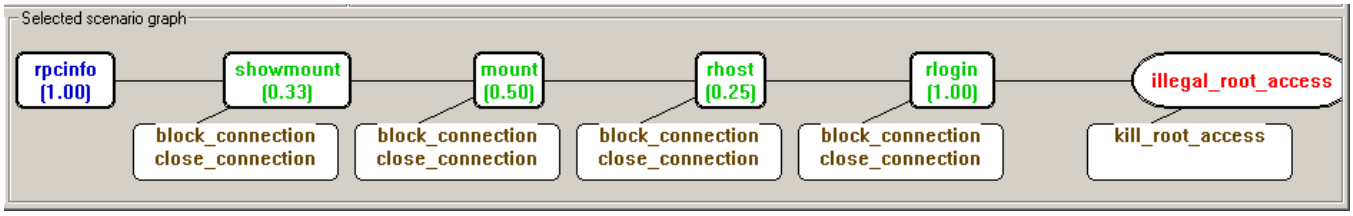

FIG. 6.6 – Résultat de la détection de la première action du scénario d'accès illégal à une machine avec les droits d'administrateur. Les réaction possibles sont affichées en dessous des hypothèses.

La première contre-mesure, *block_connection*, permet d'empêcher une machine d'accéder à une autre machine en ajoutant une règle à un firewall. La deuxième contre-mesure, *close_connection*, permet de fermer une connexion ouverte. On remarquera que les pré-condition et les post condition de ces deux modèles sont identiques étant donné que ces contre-mesures on le même effet. La dernière contre-mesure, *kill_root_access*, permet de fermer une connexion et modifie le fichier *.rhost* pour effacer les modifications permettant un accès root distant.

Utilisation d'une attaque de type SYN-flooding

Nous présentons ici un scénario d'intrusion basé sur l'inondation d'une machine par l'envoi répété de message SYN (ouverture d'une semi-connexion) sans l'envoi de message ACK (confirmation de l'ouverture d'une connexion), dans le but d'ouvrir une connexion avec une adresse forgée. Une attaque de ce type a été réalisée contre le site de Yahoo par Kevin Mitnick. Le principe du SYN-flooding est d'exploiter le fait que l'implantation de la pile TCP d'une machine détermine sa taille, et donc le nombre maximum de connexion pouvant être demandées dans un intervalle de temps réduit. Un SYN-flooding consiste donc à dépasser la capacité de la pile TCP. Le scénario que nous présentons est constitué de quatre actions. Le principe de base du scénario est d'utiliser la relation de confiance entre deux machines H_1 et H_2 d'un réseau. Cette relation est implantée par la présence dans le fichier *.rhost* de H_2 de l'adresse IP de H_1. Les quatre étapes de ce scénario sont les suivantes :

- Action 1 : *inondation de* H_1
  Cette action permet de réaliser un déni de service sur H_1. De cette manière l'attaquant peut utiliser l'adresse IP de H_1 dans des paquets forgés sans que H_1 ne puisse traiter des paquets lui étant destinés.

- Action 2 : *prédiction du numéro de séquence TCP de* H_2
  L'attaquant prédit le numéro de séquence TCP attendu par H_2 dans le prochain paquet TCP que H_2 recevra.

- Action 3 : *ouverture d'une connexion avec* H_2 *à partir de la machine de l'attaquant avec l'adresse de* H_1

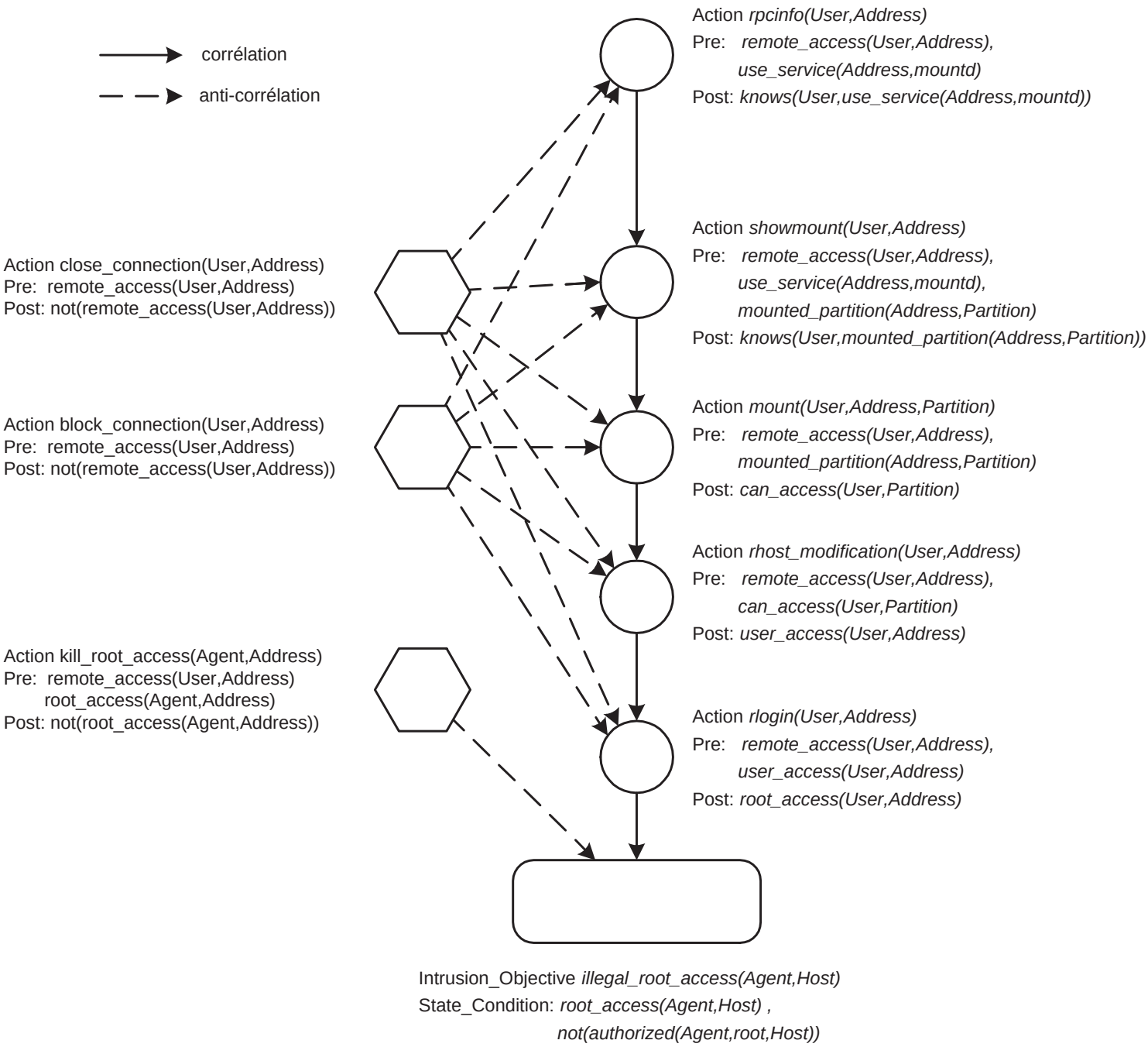

FIG. 6.7 – Graphe de corrélation du scénario d'accès illégal à une machine avec les droits d'administrateur.

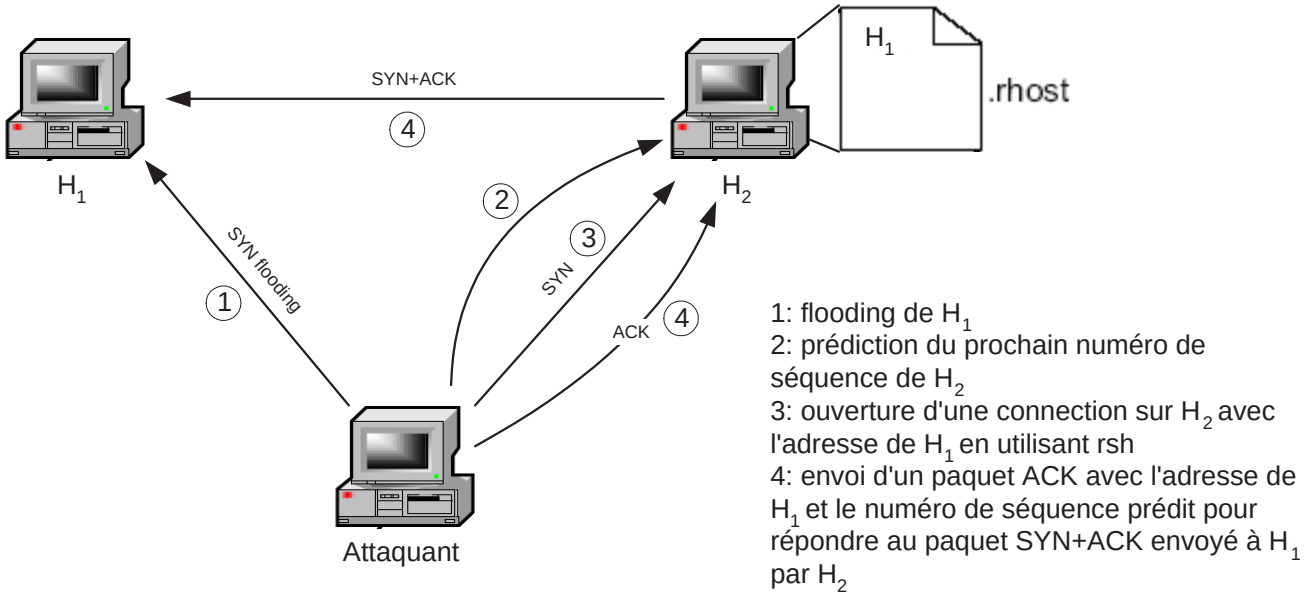

FIG. 6.8 – Principe du scénario d'attaque réalisé par Kevin Mitnick.

Cette action consiste à utiliser l'adresse de H_1, présente dans le fichier *.rhost* de H_2, pour ouvrir une connexion avec H_2. L'attaquant utilise le programe *rsh* pour ouvrir la connexion.

– Action 4 : *obtention d'un shell distant sur la machine de l'attaquant connecté à* H_2
À l'issue de la précédente action, H_2 envoie un paquet $SYN - ACK$ à H_1. H_1 ne peut répondre à ce paquet car elle est indisponible. La dernière action de l'attaquant consiste à envoyer un paquet ACK à H_2 à la place de H_1 en utilisant l'adresse de H_1 ainsi que le numéro de séquence TCP prédit grâce à la deuxième action. Ainsi l'attaquant obtient un shell distant sur H_2 avec les privilèges de H_1.

La figure 6.8 récapitule les différentes étapes de l'attaque. Après avoir obtenu le shell distant, l'attaquant peut par exemple ajouter l'adresse IP de sa machine dans le fichier *.rhost* de H_2. Ainsi l'attaquant pourra ensuite se connecter à volonté sur H_2 sans avoir à exécuter de nouveau le scénario d'attaque.

La figure 6.10 représente les modèles des actions impliquées dans ce scénario. La figure 6.11 montre le modèle de l'objectif d'intrusion associé à ce scénario. Sur la figure 6.14 nous avons représenté le graphe de corrélation du scénario. La figure 6.9 montre le résultat de la détection des deux premières actions de ce scénario par notre moteur de corrélation. Les contre-mesures utilisées pour bloquer ce scénario sont les mêmes que dans l'exemple précédent et consistent à fermer la connexion établie avec des paquets dont l'adresse est forgée. La contre-mesure s'appliquant sur l'objectif d'intrusion consiste à bloquer la connexion établie pour le shell distant. Le

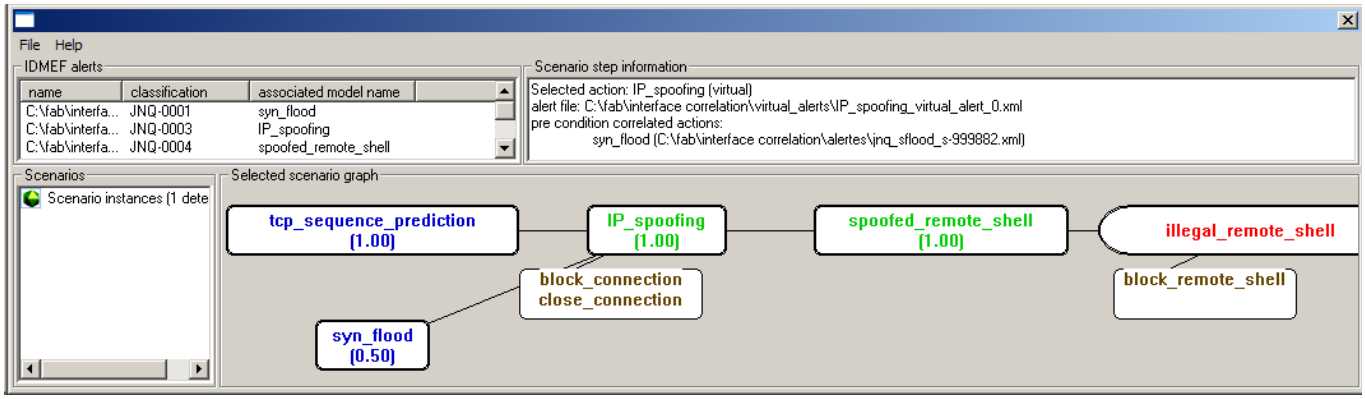

FIG. 6.9 – Résultat de la détection des deux premières actions du scénario Mitnick.

modèle de cette contre-mesure est présenté sur la figure 6.13.

L'expérimentation a permis de vérifier que le moteur de corrélation permet de bloquer l'avancement du scénario. Dans le cas où l'attaquant réussi à obtenir un shell distant, il est tout de même possible de le bloquer. Cependant il peut arriver qu'entre le moment où l'attaquant obtient le shell et le moment où la contre-mesure est lancée après confirmation par l'administrateur, l'attaquant ait pu endommager le système visé.

Déni de service distribué

Ce dernier exemple de scénario présente une attaque de type déni de service distribué (DDOS en anglais pour Distributed Denial Of Service) exécutée en utilisant l'outil *Trinoo* (voir [34] pour de plus amples informations sur cet outil). Nous allons tout d'abord exposer le fonctionnement de cet outil puis nous verrons comment nous détectons une attaque réalisée avec cet outil ainsi que les contre-mesures que nous avons développées.

Le principe de base de l'outil *Trinoo*, principe partagé par d'autres outils d'attaque en déni de service distribué (Stacheldraht, TFN, TFN2K, etc... [35, 36, 11, 81]), est d'exploiter des machines, appelées « zombies », dans lesquelles un programme esclave est installé. Dans [33], Dittrich présente un exemple de scénario permettant d'installer un outil de déni de service. Ce scénario peut être réalisé pour installer l'outil *Trinoo* :

- étape 1 : utilisation d'un compte utilisateur compromis sur une machine hébergeant un grand nombre d'utilisateurs et possédant une connexion à Internet à haut débit. Les différents exécutables nécessaires pour l'attaque sont stockés sur le compte utilisateur compromis de cette machine. Cette machine sera ensuite utilisée par l'attaquant pour communiquer avec les « zombies ».

- étape 2 : un grand nombre de réseaux sont balayés pour trouver des cibles potentielles exécutant des services exploitables à distance.

Action $syn\_flood(User, Host)$
Pre : $remote\_access(User, Host)$,
Post : $deny\_of\_service(Host)$
Detection : *classification(Alert,'SYN-flooding')*
$source(Alert, User)$
$target(Alert, Host)$

Action $tcp\_sequence\_prediction(User, Host)$
Pre : $remote\_access(User, Address)$
Post : $knows(User, tcp\_sequence(Host))$
Detection : *classification(Alert,'TCP sequence prediction')*
$source(Alert, User)$
$target(Alert, Host)$

Action $IP\_spoofing(User, Host_1, Host_2)$
Pre : $remote\_access(User, Host_2)$,
$knows(User, tcp\_sequence(Host_2))$,
$deny\_of\_service(Host_1)$
Post : $spoofed\_connection(User, Host1, Host2)$
Detection : *classification(Alert,'IP spoofing')*
$source(Alert, User)$
$target(Alert, Host2)$
$spoofed\_address(Alert, Host1)$

Action $spoofed\_remote\_shell(User, Host_1, Host_2)$
Pre : $spoofed\_connection(User, Host_1, Host_2)$
Post : $remote\_shell(User, Host_2)$
Detection : *classification(Alert,'spoofed remote shell')*
$source(Alert, User)$
$target(Alert, Host2)$
$spoofed\_address(Alert, Host1)$

FIG. 6.10 – Modèles des actions du scénario d'attaque Mitnick

Intrusion_Objective $illegal\_remote\_shell(User, Host)$
State_Condition : $remote\_shell(User, Host)$,
$\text{not}(authorized(User, remote\_shell, Host))$

FIG. 6.11 – Objectif d'intrusion visé par l'attaquant lors de l'exécution du scénario Mitnick.

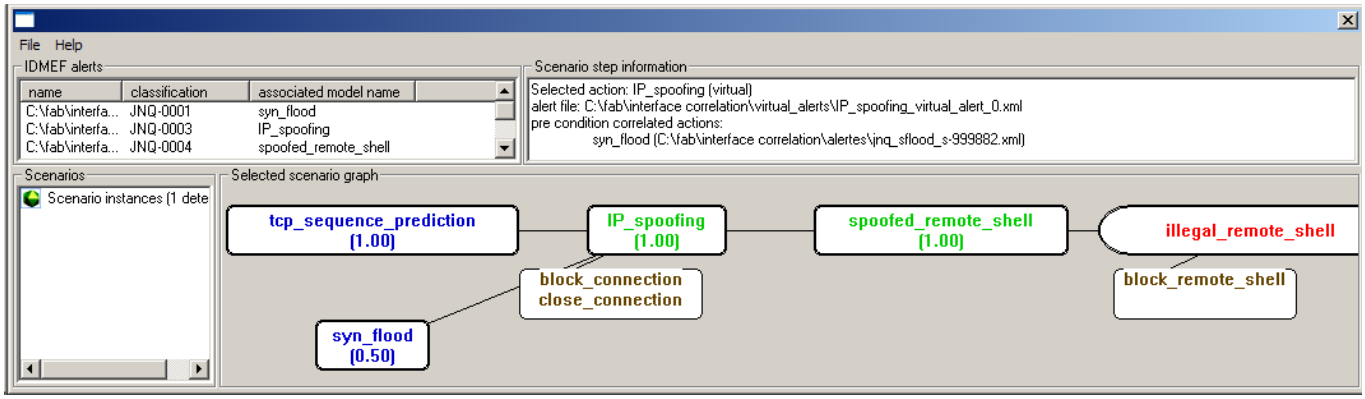

FIG. 6.12 – Détection des deux premières étapes du scénario Mitnick.

Action $kill\_remote\_shell(User, Host)$
Pre : $remote\_shell(User, Host)$
Post : $not(remote\_shell(User, Host))$

FIG. 6.13 – Modèle de la contre-mesure sur l'objectif du scénario Mitnick

– étape 3 : les machines vulnérables trouvées dans l'étape précédente sont utilisée pour installer le programme esclave les transformant en machines « zombies ».

– étape 4 : un sous ensemble des machines zombies est choisi. Si l'attaquant sait que certaines de ces machines ont un accès haut débit à Internet, il choisira sans doute ses machines.

Un script peut être utilisé pour installer l'ensemble des programmes nécessaires pour l'attaque sur un ensemble de machines. Notons que la première machine compromise durant l'étape 1 est appelée machine maître. C'est elle qui communique aux machines « zombies », appelées aussi machines esclave, les actions à exécuter. Il est aussi possible d'utiliser non pas une seule machine maître mais un ensemble de machines maître communiquant avec les machines esclave.

La figure 6.15 présente l'architecture partagée par l'ensemble des outils d'attaque en déni de service distribué. Comme nous l'avons évoqué précédemment, un ensemble de machines maître reçoit des commandes depuis la machine de l'attaquant. Ensuite, les machines maître communiquent avec les machines esclave pour leur commander l'envoi massif de trafic sur une (ou plusieurs) machine(s) cible(s) pour la(les) rendre indisponible(s).

L'outil *Trinoo* partage cette architecture et utilise plusieurs numéros de port ainsi que plusieurs protocoles pour assurer les communications entre l'attaquant et les maîtres ainsi qu'entre les maîtres et les esclaves. La figure 6.16 montre les ports et protocoles utilisés.

Le dialogue entre l'attaquant et le(s) maître(s) se fait sur le port 27665 en uti-

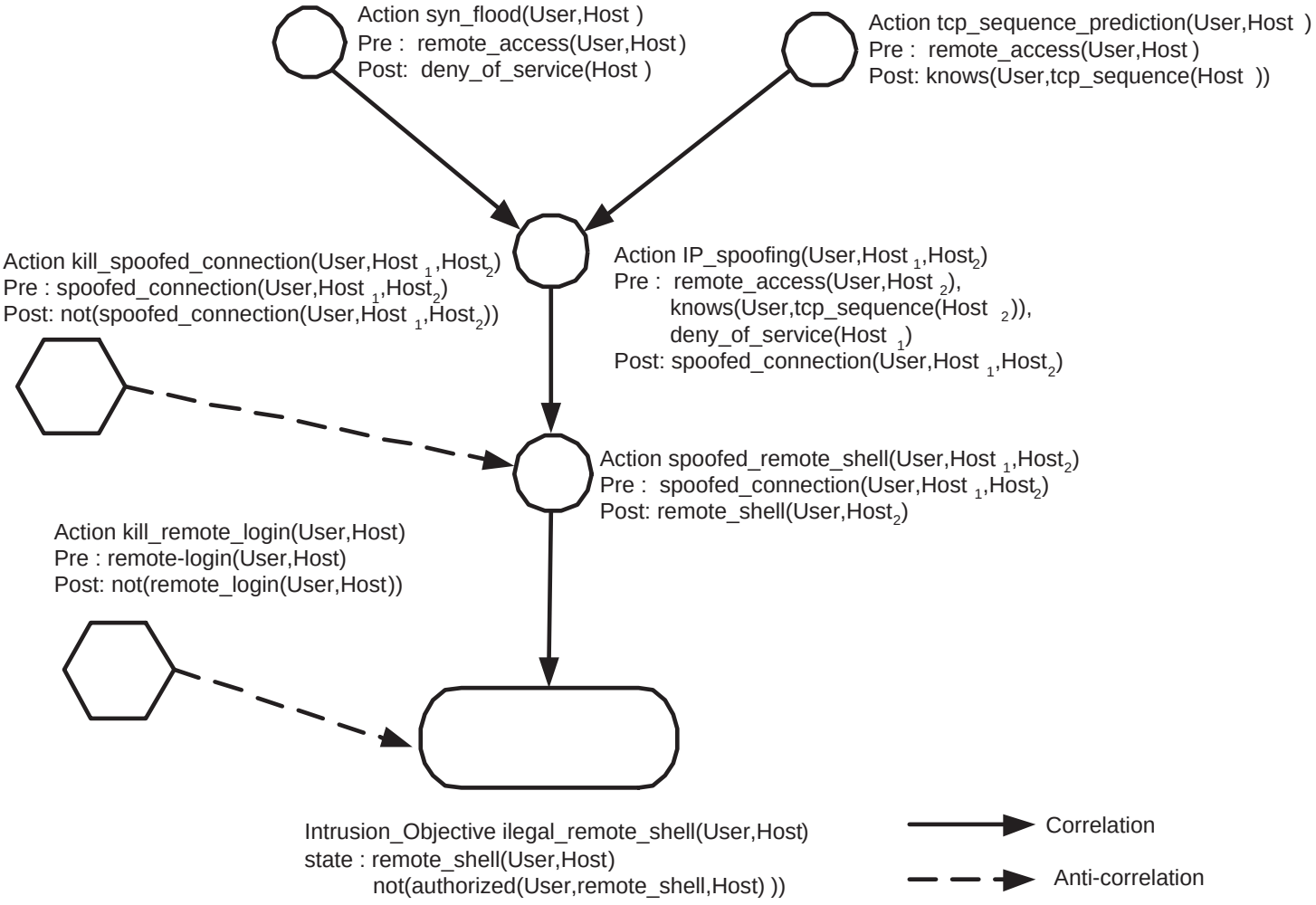

FIG. 6.14 – Graphe de corrélation du scénario Mitnick.

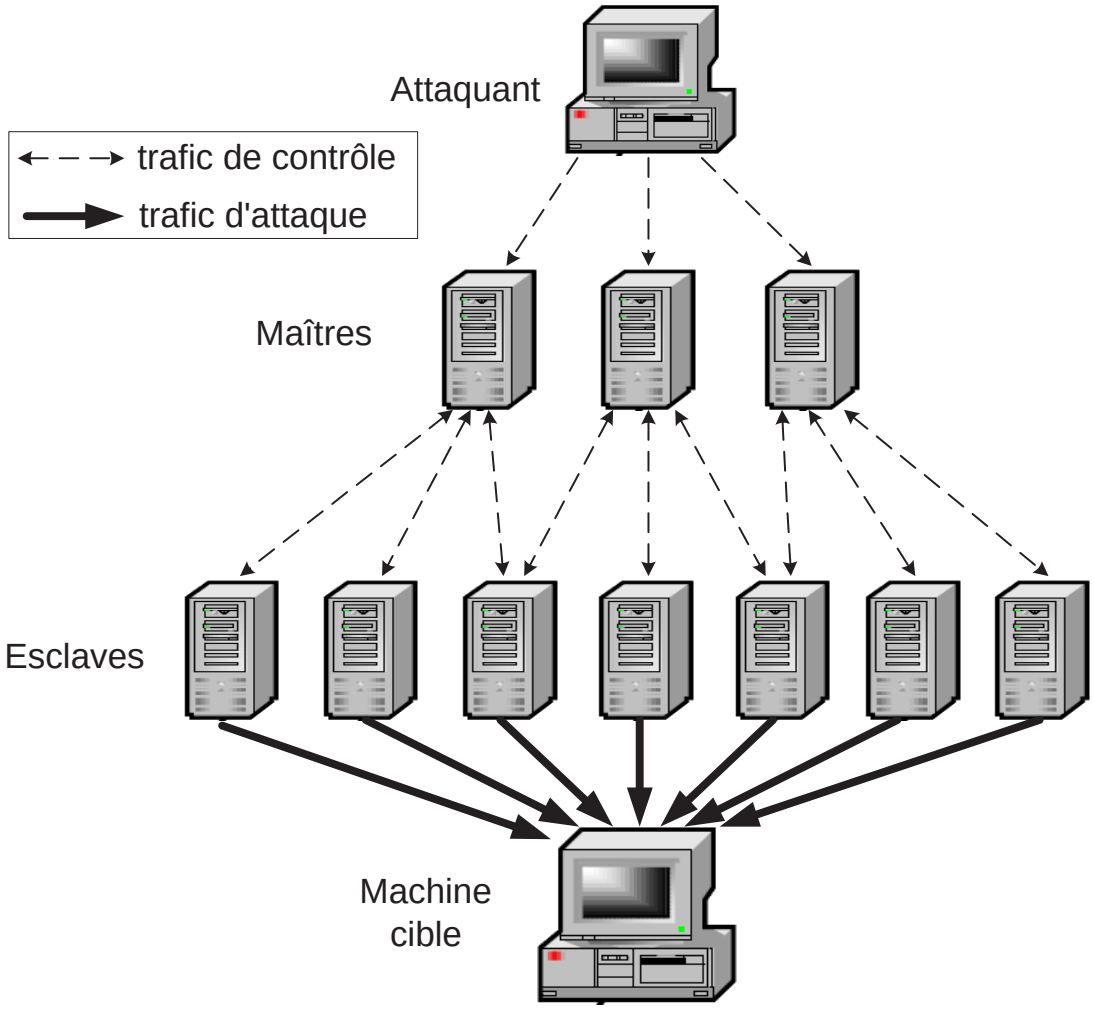

FIG. 6.15 – Architecture d'un outil d'attaque par déni de service distribué.

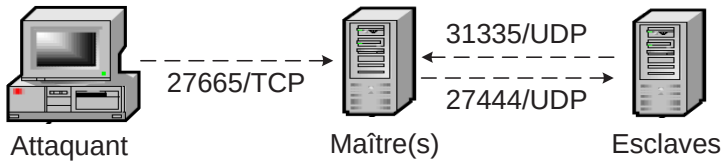

FIG. 6.16 – Numéros de port et protocoles utilisés par l'outil *Trinoo*.

lisant le protocole TCP. Il est ainsi possible d'utiliser le programme *Telnet* pour dialoguer avec le(s) maître(s) en exécutant la commande "attacker# telnet @master_host 27665". Plusieurs commandes permettent à l'attaquant de dialoguer avec le(s) maître(s), en voici quelques-unes :

- *"dos IP"* : inonde l'adresse IP spécifiée. La commande *"aaa l44adsl IP"* est envoyée par le(s) maître(s) à chacun de ses esclaves pour inonder l'adresse donnée.

- *"mdos <IP1 :IP2 : ... :IPn> "* : inonde les adresses spécifiées. Cette commande permet de réaliser un déni de service sur plusieurs machines à la fois. La commande *"xyz l44adsl 123 :IP1 :IP2 : ... :IPn"* est envoyée par le(s) maître(s) à chacun de ses esclaves pour inonder les adresses données.

D'autre part, plusieurs commandes permettent au(x) maître(s) de dialoguer avec les esclaves, en voici quelques-unes :

- *"aaa pass IP"* : à la réception de cette commande, l'esclave inonde l'adresse spécifiée avec des paquets UDP sur des ports aléatoires. *pass* est le mot de passe utilisé entre un maître et ses esclaves. Par défaut le mot de passe est *"l44adsl"*.

- *"shi pass"* : demande à l'esclave d'envoyer la chaîne de caractère $*HELLO*$ à la liste de machines maîtres sur le port 31335/UDP. La liste des maîtres associés à un esclave est compilée dans le programme.

- *"png pass"* : envoie la chaîne de caractères *PONG* à la machine maître qui a exécuté la commande sur le port 31335/UDP.

- *"d1e pass"* : ferme le programme esclave. Nous verrons plus loin que nous utilisons cette commande pour implanter une contre-mesure.

- *"xyz pass 123 :IP1 :IP2 : ... :IPn"* : a le même effet que la commande *aaa* mais sur plusieurs adresses.

Nous allons maintenant présenter comment nous modélisons une attaque réalisée avec l'outil *Trinoo* et comment nous modélisons la contre-mesure permettant d'empêcher l'attaque. Nous supposons que les différents programmes maître et esclave sont déjà installés sur les machines exploitées par un attaquant. Il serait possible de détecter l'installation de ces programmes mais seulement dans le cas où l'attaquant n'a pas d'accès physique aux machines. D'autre part, nous supposons que des sondes *Snort* sont placées sur le réseau de telle manière que les communications entre l'attaquant et le(s) maître(s) ainsi que les communications entre le(s) maître(s) et les esclaves soient détectables. Étant donné que la base de signature de *Snort* ne permet

pas de détecter ces communications, nous avons déterminé les signatures *Snort* permettant de détecter l'ensemble des commandes entre les différentes machines impliquées dans la réalisation d'un déni de service distribué.

Dans notre expérimentation, la machine de l'attaquant, le(s) machine(s) maître et les machines esclave se trouvent dans des sous-réseaux différents, hypothèse qui est généralement conforme à la réalité. Nous supposons qu'aucun SDI n'est installé dans le sous-réseau de l'attaquant car dans un cas réel il est impossible de le faire. Nous modélisons les actions suivantes :

- Action 1 : *Telnet de l'attaquant vers une machine maître*
  la connexion $Telnet$ utilisée par l'attaquant pour communiquer avec une machine maître constitue le début du scénario d'attaque.

- Action 2 : *Message « Hello » d'un esclave vers un de ses maîtres*
  ce message envoyé par un esclave vers un maître lui indique qu'il est prêt à recevoir des commandes.

- Action 3 : *Commande « dos » envoyé par l'attaquant vers une machine maître*
  cette commande envoyée par l'attaquant vers un maître lui demande d'utiliser les esclaves détectés pour commencer l'attaque distribuée.

- Action 4 : *Commande « dos » envoyée par un maître vers un esclave*
  cette commande envoyée par un maître vers un esclave lui demande de commencer à inonder la machine cible.

Les modèles de ces actions sont représentés sur la figure 6.17. L'objectif d'intrusion associé à cette attaque est représenté sur la figure 6.18. Cette attaque permet aussi d'atteindre l'objectif d'intrusion DOS_on_DNS, représenté sur la figure 4.7 de la section 4.2.3 du chapitre 4, si la machine visée par l'attaque est un DNS. Le cas d'une attaque sur un serveur DNS a été récemment rencontré dans la réalité (voir http ://seclists.org/lists/isn/2004/Jul/0106.html) ; une entreprise a vu ses serveurs DNS rendus indisponibles pendant une durée de quatre heures.

Pour empêcher l'attaquant d'exploiter les machines esclave, nous avons élaboré et modélisé une contre-mesure utilisant le protocole de dialogue de l'outil *Trinoo*. L'idée est d'arrêter les programmes esclave exécutés sur les machines esclave en utilisant la commande $d1e$ présentée plus haut. Le modèle de cette contre-mesure est représentée sur la figure 6.19. Le graphe de corrélation complet de ce scénario est représenté sur la figure 6.20. On peut voir sur ce graphe que la contre-mesure doit être lancée après que tous les esclaves soient reconnus, ou alors au fur et à mesure qu'ils sont détectés.

Le résultat de la détection des premières étapes de ce scénario sont représentés sur la figure 6.2. On peut voir que la génération d'hypothèses indique qu'un second scénario d'intrusion est possible à partir de l'action $Telnet$.

La contre-mesure consistant à fermer le programme esclave sur chaque machine

Action $telnet(A, H)$
Pre : $master(H)$,
$client\_port(27665)$
Post : $connected(A, H, 27665)$
Detection : *classification(Alert,'telnet')*
$source(Alert, A)$
$target(Alert, H)$

Action $hello(H, S)$
Pre : $master(H)$,
$slave(S)$
Post : $knows(H, slave(S))$
Detection : *classification(Alert,'hello Trinoo')*
$source(Alert, H)$
$target(Alert, S)$

Action $command\_dos(A, H, V)$
Pre : $connected(A, H, 27665)$,
$master(H)$,
$knows(H, slave(S))$
Post : $dos\_command\_sent(H, S)$
Detection : *classification(Alert,'dos cmd to master')*
$source(Alert, A)$
$target(Alert, H)$
$trinoo\_victim(Alert, V)$

Action $command\_dos\_to\_slave(H, S, V)$
Pre : $knows(H, slave(S))$,
$master(H)$,
$dos\_command\_sent(H, S)$
Post : $denial\_of\_service(V)$
Detection : *classification(Alert,'dos cmd master to slave')*
$source(Alert, H)$
$target(Alert, S)$
$trinoo\_victim(Alert, V)$

FIG. 6.17 – Modèles des actions impliquées dans la réalisation d'une attaque distribuée à l'aide de l'outil *Trinoo*.

Intrusion_Objective $denial\_of\_service(Host)$
State_Condition : $denial\_of\_service(Host)$

FIG. 6.18 – Objectif d'intrusion réalisable grâce à l'outil *Trinoo*.

Action $kill\_slave(S)$
Pre : $slave(S)$
Post : $not(slave(S))$

FIG. 6.19 – Modèle de la contre-mesure du scénario en déni de service distribué par l'outil *Trinoo*

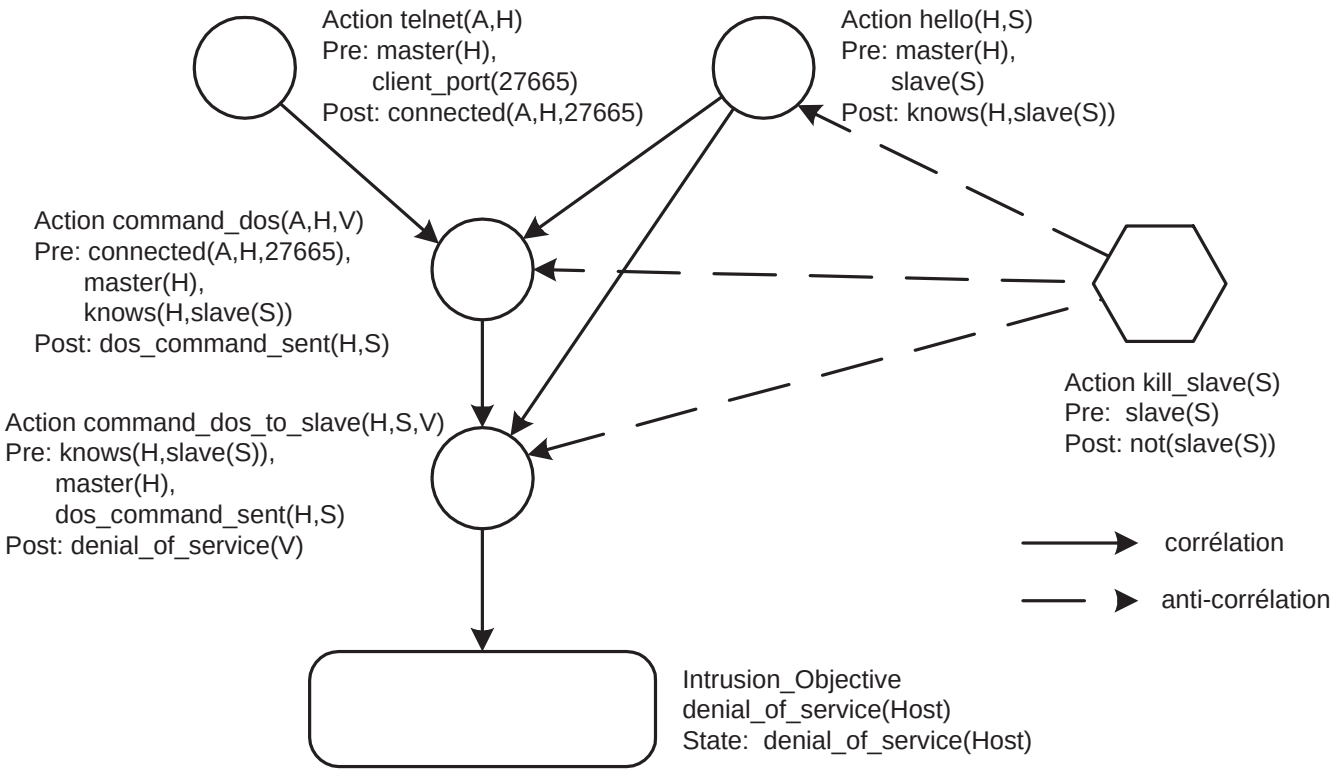

FIG. 6.20 – Graphe de corrélation du scénario en déni de service distribué par l'outil *Trinoo*.

zombie détectée n'est pas la meilleure solution. En effet si nous supposons qu'un script permettant le redémarrage automatique du programme zombie, lorsqu'il a été fermé, a été installé sur les machines esclaves, alors notre contre-mesure ne permet pas d'annuler l'attaque distribuée mais de la retarder. Une solution alternative est de bloquer le trafic UDP venant des machines infectées, mais dans ce cas il est possible que ce trafic contienne des paquets n'ayant aucun lien avec l'attaque distribuée.

6.3 Conclusion

A travers les différents scénarios que nous avons modélisé et expérimenté, nous avons montré que les différents mécanismes que nous avons définis dans les chapitres précédents nous permettent de répondre à plusieurs problèmes. Nous sommes en effet capable de détecter les scénarios, nous sommes en mesure d'anticiper la suite des scénarios à partir des premières alertes et nous pouvons contrecarrer l'avancement des scénarios grâce aux contre-mesures. D'autre part, nous fournissons à l'administrateur une représentation graphique des scénarios détectés ainsi que leurs différents poids de corrélation. Cependant, la base de données représentant la connaissance sur le système surveillé est en cours d'implantation. Nous sommes persuadés qu'une fois cette tâche achevée, d'autres résultats d'expérimentation peuvent aider à enrichir les modules de corrélation et d'anti-corrélation. Nous avons intégré le module de corrélation dans une architecture distribuée [46, 45] développée par Joaquin Garcia. Les modules de fusion/agrégation et de corrélation sont actuellement en cours d'implantation dans des versions dont le code source sera librement téléchargeable.

Chapitre 7

Conclusion

Dans ce chapitre nous concluons le travail que nous avons présenté. Nous récapitulons tout d'abord les principaux apports de notre travail puis nous terminons par les perspectives ouvertes.

7.1 Contributions

Nous avons montré pourquoi la détection d'intrusions est nécessaire pour détecter l'exploitation de failles de sécurité dans un système informatique. Nous avons vu d'autre part qu'il est nécessaire de déployer plusieurs SDI à travers l'ensemble du système surveillé pour détecter le maximum d'événements suspects. Nous avons ainsi présenté plusieurs architectures permettant de collecter l'ensemble des alertes générées et effectuant des traitements pour améliorer le diagnostic fourni.

Pour traiter le flux d'alerte, nous avons proposé une architecture centralisée faisant intervenir deux modules de traitement des alertes implantant les notions de fusion et de corrélation d'alertes. L'agrégation et la fusion d'alertes similaires, permettant de réduire la quantité d'alertes à traiter sans supprimer d'information, est réalisée grâce à une mesure de similarité. Ainsi, nous avons défini un opérateur de similarité, fonction des caractéristiques des événements associés aux alertes comparées, permettant de regrouper les alertes similaires. Nous avons ensuite montré comment les informations de ces groupes d'alertes peuvent être fusionnées pour générer des alertes dites de fusion. Grâce à cet opérateur, nous obtenons une alerte de fusion par événement détecté sur le système par l'ensemble des SDI.

Cette première étape dans le traitement des alertes n'est cependant pas suffisante pour détecter les scénarios d'intrusion constitués de plusieurs actions. En effet, nous avons présenté les travaux existants sur la notion de corrélation d'alerte, que nous avons répartis en trois approches principales : corrélation implicite, explicite et semi-explicite. Notre approche semi-explicite est basée sur un langage de modélisation, le langage LAMBDA, et consiste à trouver dans un ensemble de modèles décrits dans ce langage des liens de corrélation. Ces liens de corrélation nous permettent de trouver dans un flux d'événements des actions corrélées, une action A étant corrélée à une action B si l'exécution de A favorise celle de B. Cette approche a pour avantage

d'éviter d'avoir à exprimer une base de scénarios d'intrusion, difficile à maintenir et à rendre exhaustive.

Nous avons d'autre part présenté la génération d'hypothèses permettant d'anticiper les actions de l'attaquant en générant un ensemble de scénarios plausibles à partir des actions détectées. Ce mécanisme peut générer un grand nombre de scénarios possibles. Il est donc souhaitable d'aider l'administrateur système à déterminer le scénario le plus plausible dans cet ensemble. Pour ce faire, nous avons défini la notion de corrélation pondérée permettant de calculer une relation d'ordre sur l'ensemble des scénarios générés.

La dernière notion introduite est la notion d'anti-corrélation pondérée présentée dans le chapitre dédié à la réaction à un scénario d'intrusion. Nous avons présenté les approches existantes puis nous avons montré comment nous représentons les contre-mesures permettant de réagir à une intrusion. La notion d'anti-corrélation nous permet d'identifier des relations d'influence négative entre les contre-mesures et les actions exécutables par l'attaquant. Ainsi les contre-mesures sont modélisées de la même manière que les actions dans la corrélation. Cette approche a pour avantage de ne pas figer l'ensemble des réponses possibles à une action faisant partie d'un scénario d'intrusion. En effet, les contre-mesures applicables sur une action n'ont pas à être associées manuellement. Les liens d'influence négative entre contre-mesures et actions sont trouvés automatiquement à partir des modèles.

Nous avons finalement montré les applications des notions de fusion et de corrélation/anti-corrélation au travers des expérimentations que nous avons réalisées.

7.2 Perspectives

Les perspectives ouvertes par ce travail sont nombreuses. Par exemple en ce qui concerne la notion de similarité entre alerte, on peut penser l'utiliser pour fusionner des clusters d'alertes. Par exemple, en utilisant une hiérarchie de modules d'agrégation et de fusion (à la manière d'EMERALD, voir section 2.2.2), on peut fusionner des événements de granularité variable sur le temps. On peut penser à l'agrégation d'alertes générées par des problèmes de configuration comme présenté dans [59]. Considérons par exemple qu'un routeur connaisse une panne périodique liée à un problème de configuration du système. Cette panne peut générer par exemple des ensembles de quelques centaines d'alertes à chaque panne. Si nous n'utilisons qu'un seul module de fusion pour traiter ce problème, nous sommes obligés d'agréger les alertes sur une grande période de temps en relaxant l'influence de la similarité sur les attributs temporels. En faisant cela nous risquons d'agréger des alertes qui ne sont pas liées au problème de configuration. Ainsi, il est préférable d'agréger des clusters d'alertes pour réunir des clusters éloignés dans le temps mais relatifs au même problème.

Nous avons évoqué la nécessité de représenter la connaissance sur l'état du système surveillé pour pouvoir évaluer les variables d'un modèle d'action associé à une alerte. Cette connaissance est par exemple nécessaire pour pouvoir éliminer certains faux positifs représentant des attaques exécutées sur des machines non

vulnérables. Le travail de la modélisation des propriétés d'un système informatique a été abordé dans [71]. Il reste à définir comment alimenter une base de données s'appuyant sur un modèle de cette connaissance. On peut penser à l'utilisation d'outils passifs ou actifs permettant de découvrir les propriétés du système pour alimenter cette base de données et la maintenir à jour.

Dans le chapitre dédié à la réaction aux intrusions, nous avons proposé la notion d'anti-corrélation pondérée pour sélectionner la contre-mesure la plus adaptée à une action faisant partie d'un scénario d'intrusion. Nous avons montré que la réaction à un objectif d'intrusion diffère de la réaction à une action faisant partie d'un scénario d'intrusions. En effet, un objectif d'intrusion étant modélisé par une condition sur l'état du système, il est nécessaire de nier les prédicats de cette condition pour invalider l'objectif. Les contre-mesures que nous avons présentées ne modifient pas la politique de sécurité, sauf dans le cas où la connexion de l'attaquant est empêchée en ajoutant une règle dans un firewall du système (exemple de la section6.2.1). La mise à jour de la politique de sécurité dans le cadre de la réaction à une intrusion constitue un axe de recherche à approfondir, pouvant peut-être déboucher sur une réaction totalement automatique. Plusieurs questions restent ouvertes : comment prendre en compte tous les effets des contre-mesures sur le système ? C'est-à-dire comment prendre en compte les effets d'une contre-mesure C représentée par les prédicats de $post(C)$ non anti-corrélés avec l'hypothèse ou l'objectif visé ? Comment choisir les contre-mesures permettant de restaurer l'état du système après une attaque réussie ?

Les fonctionnalités de CRIM intéressent les acteurs du monde du logiciel professionnel de détection d'intrusion. Ainsi EADS et l'entreprise Exaprotect sont intéressés par l'intégration des fonctionnalités de CRIM dans leur outils.

Comme nous l'avons précisé en conclusion du chapitre 6, le développement de CRIM continue et sera bientôt disponible en tant que logiciel libre. Le développement des nouvelles versions des modules de CRIM s'est fait dans le cadre du projet RNTL *DICO* (Détection d'intrusions COopérative)[28].

Bibliographie

[1] Advanced dragon intrusion detection system. http ://www.enterasys.com/training/certification/courses/ES-ADV-Dragon.html.

[2] Cisco ids. http ://www.cisco.com/warp/public/cc/pd/sqsw/sqidsz/.

[3] prelude hybrid ids. http ://www.prelude-ids.org.

[4] Realsecure network. http ://www.iss.net.

[5] Sentivist ids. http ://www.nfr.com/solutions/sentivist-ids.php.

[6] Shadow project. http ://www.nswc.navy.mil/ISSEC/CID/.

[7] The simplest online database that could possibly work. http ://wiki.org.

[8] J.P. Anderson. Computer security threat monitoring and surveillance. Technical report, James P. Anderson Co., Fort Washington, Pennsylvania, 1980.

[9] D. Aucutt. Hummingbird communication protocol, technical report, 1994.

[10] I. Balepin, S. Maltsev, J. Rowe, and K. Levitt. Using specification-based intrusion detection for automated response. In *Proceedings of RAID*, 2003.

[11] J. Barlow and W. Thrower. TFN2K - An Analysis. Available at : http ://packetstormsecurity.com/distributed/TFN2k_Analysis-1.3.txt, March 2000.

[12] S. Benferhat, F. Autrel, and F. Cuppens. Enhanced correlation in a intrusion detection process. In *In* Mathematical Methods, Models and Architecture for Computer Network Security (MMM-ACNS 2003)*, St Petersburg, Russia, September 2003*, 2003.

[13] A. Borgida, Ronald J. Brachman, Deborah L. McGuinness, and L. Alperin Resnick. CLASSIC : a structural data model for objects. In *ACM SIGMOD Record*, pages 58–67, 1989.

[14] R. J. Brachman, D. L. McGuinness, P. F. Patel-Schneider, L. A. Resnick, and A. Borgida. Living with classic : When and how to use a kl-one-like language. In *Principles of Semantic Networks*, pages 401–456, 1991.

[15] D. Bruschi and E. Rosti. Angel : a tool to disarm computer systems. In *NSPW'OI, September 10-13 'h, Cloudcmfl, New Mexico, USA.*, 2002.

[16] J. Buhmann. Learning and data clustering. In *Arbib, M. (Eds.), Handbook of Brain Theory and Neural Networks*, 1995.

[17] S. Cheung, U. Lindqvist, and Martin W. Fong. Modeling multistep cyber attacks for scenario recognition. Available at : citeseer.ist.psu.edu/cheung03modeling.html, 2003.

[18] F. Cuppens. Managing alerts in a multi-intrusion detection environment. In *17th Annual Computer Security Applications Conference (ACSAC'01)*, 2001.

[19] F. Cuppens, F. Autrel, Y. Bouzida, J. Garcia, S. Gombault, and T. Sans. Selecting appropriate counter-measures in a intrusion detection framework. *Annales des télécommunications*, 2005.

[20] F. Cuppens, S. Benferhat, F. Autrel, and A. Miège. Recognizing malicious intention in an intrusion detection process. In *Special session : Hybrid Intelligent Systems for Intrusion Detection, In* Second International Conference on Hybrid Intelligent Systems (HIS'2002)*, vol. 87, pp. 806-817, Santiago, Chile*, October 2002.

[21] F. Cuppens, N. Cuppens-Boulahia, T. Sans, and A. Miège. A formal approach to specify and deploy a network security policy. In *Proceedings of FAST*, 2004.

[22] F. Cuppens, S. Gombault, and T. Sans. Selecting appropriate counter-measures in an intrusion detection framework. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop*, 2003.

[23] F. Cuppens and A. Miège. Alert correlation in a cooperative intrusion detection framework. In *In Proceedings of the IEEE Symposium on Research in Security and Privacy, pp. 202-215, Oakland, USA*, May 2002.

[24] F. Cuppens and R. Ortalo. Lambda : a language to model a database for detection of attacks. In *Recent Advances in Intrusion Detection, Proc. 4th Int'l Symp., RAID*, 2001.

[25] F. Cuppens, T. Sans, and S. Gombault. Selecting appropriate counter-measures in an intrusion detection framework. In *proceedings of the 17th IEEE Computer Security Foundations Workshop*, 2004.

[26] H. Debar. Détection d'intrusions. vers un usage réel des alertes. In *Université de Caen. Habilitation à Diriger des Recherches*, 2004.

[27] H. Debar and D. Curry. The IDMEF format. Available at : http ://www.ietf.org/html.charters/idwg-charter.html, 2004.

[28] H. Debar, B. Morin, F. Cuppens, F. Autrel, L. Mé, B. Vivinis, S. Benferhat, M. Ducasse, and R. Ortalo. Détection d'intrusions : corrélation d'alertes. *Revue TSI 2004 - page n° 359*, 2004.

[29] H. Debar and A. Wespi. Aggregation and correlation of intrusion-detection alerts. In *Recent Advances in Intrusion Detection, Proc. 4th Int'l Symp., RAID*, 2001.

[30] R. Deraison, M. Arboi, and J. Lampe. Nessus security scanner. http ://www.nessus.org/index.html.

[31] L. Deri, S. Suin, and G. Maselli. Design and implementation of an anomaly detection system : An empirical approach, 2003.

[32] Luca Deri. Free network traffic probe. http ://www.ntop.org/ntop.html.

[33] D. Dittrich. Distributed Denial of Service (DDoS) Attacks and Tools. Available at : http ://staff.washington.edu/dittrich/misc/ddos/.

[34] D. Dittrich. The DoS Project's "trinoo" distributed denial of service attack tool. Available at : http ://staff.washington.edu/dittrich/misc/trinoo.analysis.txt, October 1999.

[35] D. Dittrich. The "stacheldraht" distributed denial of service attack tool. Available at : http ://staff.washington.edu/dittrich/misc/stacheldraht.analysis.txt, December 1999.

[36] D. Dittrich. The "Tribe Flood Network" distributed denial of service attack tool. Available at : http ://staff.washington.edu/dittrich/misc/tfn.analysis.txt, October 1999.

[37] C. Dousson. Suivi d'évolutions et reconnaissance de chroniques. http ://dli.rd.francetelecom.fr/abc/diagnostic/, 1994.

[38] S. Eckmann, G. Vigna, and R. Kemmerer. Statl : An attack language for state-based intrusion detection. Dept. of Computer Science, University of California, Santa Barbara., 2000.

[39] J. Evans. Hummingbird trust system, technical report, 1995.

[40] H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information, May 2003.

[41] David F. Ferraiolo, John F. Barkley, and D. Richard Kuhn. A role-based access control model and reference implementation within a corporate intranet. *ACM Transactions on Information and System Security*, 2(1) :34–64, 1999.

[42] International Organisation for Standardisation. Information processing systems – open systems interconnection – basic reference model – part 2 : Security architecture. http ://www.iso.org/.

[43] D. Frincke, D. Tobin, J. McConnell, J. Marconi, and D. Polla. A framework for cooperative intrusion detection. In *NIST National Information Systems Security Conference*, 1998.

[44] Fyodor. Nmap free security scanner. http ://www.insecure.org/nmap/.

[45] J. Garcia, F. Autrel, J. Borrel, S. Castillo, F. Cuppens, and G. Navarro. Preventing coordinated attacks via alert correlation. In *In Proceedings of the Ninth Nordic Workshop on Secure IT Systems Encouraging Cooperation (NORDSEC)*, November 2004.

[46] J. Garcia, F. Autrel, J. Borrel, S. Castillo, F. Cuppens, and G. Navarro. Decentralized publish-subscribe system to prevent coordinated attacks via alert correlation. In *In Proceedings of the Sixth International Conference on Information and Communications Security (ICICS'04), number 3269 in LNCS*, October 2004.

[47] Robert P. Goldman, W. Heimerdinger, Steven A. Harp, Christopher W. Geib, V. Thomas, and Robert L. Carter. Information modeling for intrusion report aggregation. In *DISCEX*, 2001.

[48] R. Gopalakrishna. A framework for distributed intrusion detection using interest driven cooperating agents. *Paper for Qualifier II examination, Department of Computer Sciences, Purdue University*, 2001.

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

[50] J. Hartigan. Clustering algorithms. New York : Wiley, 1975.

[51] A. Hinneburg and Daniel A. Keim. Optimal grid-clustering : Towards breaking the curse of dimensionality in high-dimensional clustering. In *The VLDB Journal*, pages 506–517, 1999.

[52] Ming-Yuh Huang, R.J. Jasper, and Thomas M. Wicks. A large scale distributed intrusion detection framework based on attack strategy analysis. *Computer Networks (Amsterdam, Netherlands)*, 31(23–24) :2465–2475, 1999.

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

[54] J. Ingalls, T. Lawrence, E. Lustig, and Hummingbird Team B. Technical report, 1997.

[55] R. Jagannathan, T. Lunt, D. Anderson, C. Dodd, F. Gilham, C. Jalali, H. Javitz, P. Neumann, A. Tamaru, and A. Valdes. Next-generation intrusion detection expert system (nides).

[56] A.K. Jain and R.C. Dubes. Algorithms for clustering data. Prentice Hall, 1988.

[57] S. Javitz and A. Valdes. The sri ides statistical anomaly detector. In *IEEE Symposium on Security and Privacy*, Menlo Park, CA 94025, may 22-23, 1991.

[58] S. C. Johnson. Hierarchical clustering schemes. In *Psychometrika*, pages 241–254, 1967.

[59] K. Julisch. Using root cause analysis to handle intrusion detection alarms. Phd Thesis, 2003.

[60] Richard Y. Kain and Carl E. Landwehr. On access checking in capability-based systems. In *IEEE Symposium on Security and Privacy*, pages 95–101, 1986.

[61] A.Abou El Kalam, R. El-Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miege, C. Saurel, and G. Trouessin. Orbac : un modèle de contrôle d'accès basé sur les organisations. In *Cahiers Francophones de la Recherche en Sécurité de l'Information*, pages 30–40, 2003.

[62] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium San Francisco, California, USA August 5-9*, 2002.

[63] Network Associates Laboratories. Secure execution environments/generic software wrappers for security and reliability. http ://www.networkassociates.com.

[64] U. Lindqvist and Phillip A. Porras. Detecting computer and network misuse through the production-based expert system toolset (p-BEST). In *IEEE Symposium on Security and Privacy*, pages 146–161, 1999.

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

[66] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

[67] M. Mahoney. Network traffic anomaly detection based on packet bytes. In *In Proc. ACM-SAC*, 2003.

[68] J. McDermott. Attack net penetration testing. In *The New Security Paradigms Workshop*, pages 15–22, 2000.

[69] C. Michel and L. Mé. Adele : an attack description language for knowledge-based intrusion detection. In *Oakland*, 2001.

[70] B. Morin and H. Debar. Correlation of intrusion symptoms : an application of chronicles. In *Proceedings of the 6th symposium on Recent Advances in Intrusion Detection (RAID)*, 2003.

[71] B. Morin, L. Mé, H. Debar, and M. Ducassé. M2d2 : A formal data model for ids alert correlation. In *RAID*, pages 115–127, 2002.

[72] H. Moulin. Axioms of cooperative decision making. Cambridge University Press, Cambridge, 1988.

[73] P. Neisen, C. Coltrin, J. Dowding, and Hummingbird Team A. Technical report, 1997.

[74] B. Neuman and T. Ts'o. Kerberos : An authentication service for computer networks. http ://nii.isi.edu/publications/kerberos-neuman-tso.html.

[75] P. Ning, Y. Cui, and D. Reeves. Constructing attack scenarios through correlation of intrusion alerts, November 2002.

[76] P. Porras and P. Neumann. Emerald : Event monitoring enabling responses to anomalous live disturbances. In *Proceedings of the National Computer Security Systems Conference, pages 719-729*, 1997.

[77] Jean-Philippe Pouzol and Mireille Ducassé. From declarative signatures to misuse IDS. *Lecture Notes in Computer Science*, 2212 :1–21, 2001.

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

[79] R. Sandhu. Access control : The neglected frontier. In *First Australian Conference on Information Security and Privacy*, Wollong, Australia, 1996.

[80] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2) :38–47, 1996.

[81] Packetstorm Security. Distributed Denail oF Service Attacks Tools. Available at : http ://packetstormsecurity.com/distributed/, 2000.

[82] M. Roesch. Snort. lightweight intrusion detection for networks. In *In Proc. of LISA*, 1999.

[83] J. Steffan and M. Schumacher. Collaborative attack modeling. In *Proceedings of SAC*, 2002.

[84] S. Templeton and K. Levitt. A requires/provide model for computer attacks. In *Proceedings of the ACM New Security Paradigms Workshop*, pages 31–38, 2000.

[85] Miika Turkia. Intrusion detection systems. citeseer.ist.psu.edu/435495.html.

[86] A. Valdes and K. Skinner. Probabilistic alert correlation. *Lecture Notes in Computer Science*, 2212 :54–68, 2001.

[87] Paul A. Watson. Slipping in the window : Tcp reset attacks. http ://www.osvdb.org/reference/SlippingInTheWindow_v1.0.doc.

Annexe A

Articles publiés

F. Cuppens, F. Autrel, A. Miège et S. Benferhat. Correlation in an intrusion detection process. Internet Security Communication Workshop (SECI'02), Tunis, Septembre 2002.

F. Cuppens, F. Autrel, A. Miège and S. Benferhat, Recognizing malicious intention in an intrusion detection process, Special session : "Hybrid Intelligent Systems for Intrusion Detection", In *Second International Conference on Hybrid Intelligent Systems (HIS'2002)*, vol. 87, pp. 806-817, Santiago, Chile, October 2002.

S. Benferhat, F. Autrel et F. Cuppens, Weighted Correlation in an Intrusion Detection Process. 2ème rencontre francophone sur Sécurité et Architecture Réseaux (SAR'2003), Nancy, France, 30 Juin-4 Juillet, 2003.

S. Benferhat, F. Autrel and F. Cuppens, Enhanced correlation in a intrusion detection process, In *Mathematical Methods, Models and Architecture for Computer Network Security (MMM-ACNS 2003)*, St Petersburg, Russia, September 2003.

H. Debar, B. Morin, F. Cuppens, F. Autrel, L. Mé, B. Vivinis, S. Benferhat, M. Ducasse and R. Ortalo, Détection d'intrusions : corrélation d'alertes, *Revue TSI 2004* - page n° 359.

J. Garcia, F. Autrel, J. Borrell, S. Castillo, F. Cuppens and G. Navarro, Decentralized publish-subscribe system to prevent coordinated attacks via alert correlation, In *Proceedings of the Sixth International Conference on Information and Communications Security (ICICS'2004)*, number 3269 in LNCS, October 2004.

J. Garcia, F. Autrel, J. Borrel, S. Castillo, F. Cuppens and G. Navarro, Preventing coordinated attacks via alert correlation, In Proceedings of the Ninth Nordic Workshop on Secure IT Systems Encouraging Cooperation (NORDSEC'2004), November 2004.

F. Autrel, S. Benferhat et F. Cuppens. Utilisation de la corrélation pondérée dans

un processus de détection d'intrusion. Annales des Télécommunications, éditions Hermes. 2004.

F. Cuppens, F. Autrel, Y. Bouzida, J. Garcia, S. Gombault and T. Sans, Selecting appropriate counter-measures in a intrusion detection framework, Annales des Télécommunications, éditions Hermes. 2004.

Fusion, corrélation pondérée et réaction dans un environnement de détection d'intrusions coopérative

Résumé : Les systèmes informatiques doivent respecter certaines propriétés telles que la confidentialité, l'intégrité et la disponibilité. Cependant, il existe des vulnérabilités qui permettent de violer la politique de sécurité. La détection d'intrusions a pour but de détecter l'exploitation de ces vulnérabilités. L'approche consistant à faire coopérer plusieurs sondes de détection d'intrusions permet d'améliorer le diagnostic fournit. Cette thèse développe les notions de fusion, corrélation pondérée et réaction. La fusion d'alerte regroupe les alertes redondantes pour les fusionner. La corrélation pondérée identifie des scénarios d'intrusions et sélectionne le plus plausible. La réaction bloque un scénario d'intrusions en cours d'exécution ou modifie l'état du système pour éliminer une vulnérabilité ou compenser les effets d'une attaque. Des résultats expérimentaux obtenus sur plusieurs scénarios d'intrusions à partir d'un prototype implantant les notions développées sont présentés.

Mots clés : détection d'intrusion coopérative - agrégation - fusion - corrélation pondérée - réaction

Fusion, weighted correlation and reaction in a cooperative intrusion detection context

Abstract : Computer systems must respect some properties such as confidentiality, integrity and availability. However, vulnerabilities exist and allow to violate the security policy. Intrusion detection aims at detecting the use of those vulnerabilities. Adopting a cooperative approach that uses mutliples probes gives a better diagnosis. This thesis develop the notions of fusion, weighted correlation and reaction. Alert fusion groups redundant alerts to fuse their informations. Weighted correlation finds intrusion scenarios in the alert stream and selects the most plausible one. the reaction blocks an ongoing scenario or modify the system'state to correct a vulnerability or to compensate the effects of an attack. Experimental results are given and show the use of our prototype, that implements those three notions, to detect several intrusion scenarios.

Keywords : coopérative intrusion detection - aggregation - fusion - weighted correlation - reaction