Discus : Une architecture de détection d'intrusions réseau distribuée basée sur un langage dédié

Damien Riquet

Discus : Une architecture de détection d'intrusions réseau distribuée basée sur un langage dédié

Mémoire pour l'obtention du titre de

Docteur de l'Université Lille 1

Discipline : Informatique

par

Damien Riquet

présenté le

3 décembre 2015

Composition du jury :

Président :	Lionel Seinturier	*Professeur - Université Lille 1 - CRIStAL*
Rapporteurs :	Christine Morin	*Directeur de Recherche - Inria Rennes Bretagne Atlantique - IRISA*
	Laurent Réveillère	*Maître de conférences HDR - Bordeaux INP - LaBRI*
Examinateurs :	Damien Deville	*Responsable R&D - Stormshield*
	Pierre Paradinas	*Professeur titulaire de chaire - CNAM*
Directeur :	Gilles Grimaud	*Professeur - Université Lille 1 - CRIStAL*
Encadrant :	Michaël Hauspie	*Maître de conférences HDR- Université Lille 1 - CRIStAL*

Numéro d'ordre : 41886

Table des matières

1

Introduction

1.1 Contexte

Depuis de nombreuses années, les systèmes informatiques sont omniprésents dans nos vies et la sécurisation de ces équipements est primordiale. La démultiplication d'infrastructures larges et complexes utilisées par le grand public, telles que le Cloud Computing, renforce les enjeux de la sécurité. En effet, il est nécessaire de fournir des services fiables et des mesures qui assurent la confidentialité, l'intégrité et la disponibilité de ces services.

La sécurisation de telles architectures soulève un grand nombre de problématiques, que ce soit au niveau des systèmes déployés ou des moyens de communications employés entre ces équipements. L'obtention d'un système complètement sécurisé et hermétique aux attaques relève du domaine de l'utopie, étant donné le fourmillement constant dans le domaine de la sécurité. Effectivement, des failles de sécurité sont révélées chaque jour et les équipes de développement essayent de tenir la cadence pour proposer un service fiable. La seule solution pour ce jeu du chat et de la souris est de s'appuyer sur une méthodologie permettant de rapidement pouvoir comprendre la faille, développer un correctif et le déployer sur le parc de machines concerné.

Nos travaux s'intéressent à cette problématique, plus particulièrement à la sécurité des réseaux de communications entre ces systèmes. L'interconnexion grandissante de ces infrastructures complexes et partagées est la source de nouveaux verrous scientifiques et technologiques, tels que la collaboration des solutions de sécurité, la configuration et la mise à jour des règles définissant la politique de sécurité, ou encore le passage à l'échelle de la sécurité.

1.2 Thèse

Le Cloud Computing est une architecture populaire apparue récemment, proposant des services qui s'adaptent à la demande de l'utilisateur. Nos travaux s'intéressent à ce type de structures, constitué d'un grand nombre de serveurs et d'équipements réseau, ainsi qu'aux réseaux plus conventionnels. La sécurité de ces structures est assurée par un certain nombre d'équipements, tel que les pare-feu ou les systèmes de détection d'intrusions, qui analysent le trafic réseau, détectent les attaques et réagissent en conséquence pour maintenir un certain niveau de qualité de service.

Initialement conçues et développées pour des réseaux traditionnels, ces solutions de sécurité ne s'adaptent pas à des structures complexes et larges comme le Cloud Computing. En effet, ces solutions partent généralement du constat qu'elles protègent le réseau intérieur des menaces provenant du réseau externe. Cette approche est désormais dépassée avec le Cloud, où tout un chacun peut louer des services puissants, pour une durée spécifique. La menace peut donc maintenant provenir de l'intérieur du réseau, quand un utilisateur mal intentionné décide d'utiliser les services du Cloud pour mener une attaque. Considérer que les attaques peuvent provenir de l'intérieur du réseau ne suffira pas à sécuriser le système. En effet, pour détecter une attaque, une solution de sécurité a besoin d'analyser des événements relatifs à cette attaque. L'attaque

ne transitant pas nécessairement par une solution de sécurité, elle peut ne pas être détectée.

Nous proposons dans nos travaux la solution de sécurité nommée Discus, qui s'appuie sur les solutions de sécurité existantes. En plus de celles-ci, nous déployons un réseau de sondes de sécurité distribué sur le réseau, permettant de conduire des analyses sur l'ensemble du réseau. Ce réseau est constitué de sondes hétérogènes, qui peuvent être physiquement ajoutées sur le réseau ou virtuellement installées sur les postes à surveiller. Ces sondes communiquent entre elles pour détecter collaborativement les intrusions au travers d'un mécanisme de base de données distribuée. Cette architecture soulève deux axes de problématique. Le premier concerne la configuration de ce parc de sondes. Il n'est pas envisageable de confier à une personne la tâche de configurer ces sondes, qui sont hétéroclites et dont les capacités ou modes de fonctionnement sont spécifiques. Le deuxième axe se concentre sur les problématiques de distribution : établir la nature des objets à distribuer, les mécanismes de distribution efficaces et les moyens de déterminer qui distribue quel élément.

Nos travaux s'orientent principalement sur le premier axe de problématique, à savoir la configuration de l'ensemble de ces sondes. Étant donné que ces sondes présentent chacune des caractéristiques spécifiques (en terme de mémoire, de rapidité d'analyse, etc.) et qu'elles se situent en un point précis du réseau, il est nécessaire de proposer une configuration unique à chacune d'entre elles. Il n'est cependant pas envisageable qu'une personne ait l'expertise sur l'ensemble des domaines en jeu (sécurité, développement embarqué, administration réseau, etc.) et soit capable de développer pour chacune des sondes une configuration spécifique. C'est pourquoi nos travaux se sont orientés vers l'élaboration d'un langage dédié, Discus Script, qui propose une couche d'abstraction. Notre langage permet de se concentrer sur la conception d'une politique de sécurité plutôt que sur des problématiques spécifiques aux sondes déployées, qui seront gérées par des experts du domaine. Nous présentons dans ce mémoire le langage Discus Script, ainsi qu'une évaluation comparative, examinant l'expressivité, la robustesse et les performances de notre langage.

1.3 Structure du document

Cette thèse se décompose en six chapitres, incluant cette introduction.

Le **chapitre 2** présente un état de l'art consacré aux thèmes abordés durant la thèse : les architectures réseaux larges telles que le Cloud Computing, la sécurité réseau et leurs méthodes de détection, ainsi que les langages dédiés à la sécurité réseau.

Le **chapitre 3** présente le contexte de ces travaux et discute des solutions existantes dans ce cadre. Puis il met en lumière la problématique étudiée dans ce mémoire, appuyée par des expérimentations. Enfin, la solution Discus (sur laquelle ces travaux se basent) est présentée.

Le **chapitre 4** présente Discus Script, le langage dédié qui accompagne la solution présentée dans le chapitre 3. Il décrit la philosophie et les éléments de syntaxe du langage. De plus,

des cas d'utilisation de ce langage sont présentés au travers d'exemples.

Le **chapitre 5** présente les propriétés du langage Discus Script et discute des avantages qu'il apporte par rapport aux autres langages existants. Il se focalise notamment sur la propriété d'expressivité du langage et sur sa robustesse. Ce chapitre présente également des expérimentations comparant l'efficacité du code généré par rapport aux solutions existantes, en terme d'empreinte mémoire, de taux de détection et de temps de traitement.

Le **chapitre 6** conclut ces travaux, discute de leurs limites, puis ouvre sur leurs perspectives.

2

État de l'art

L'émergence de nouvelles architectures, telles que le Cloud Computing, pose un ensemble de nouvelles problématiques, que ce soit au niveau de la distribution des données ou des tâches, de la qualité de service ou encore de la sécurité des services proposés. Nos travaux s'intéressent à cette dernière. Dans ce chapitre, nous présentons et discutons tout d'abord des solutions existantes de sécurité réseau. Ensuite, nous discutons des conséquences de l'émergence de ces nouvelles infrastructures. Pour finir, nous présentons les travaux relatifs aux langages dédiés à la description de règles de sécurité réseau et leur application dans le contexte de larges centres de calcul mutualisés.

2.1 Sécurité réseau

Dans cette section, nous présentons les éléments existants dans la littérature relatifs à la sécurité réseau. Pour cela, nous commençons par définir ce que l'on entend par sécurité réseau. Puis nous décrivons les solutions de sécurité existantes. Ensuite, nous nous focalisons sur les Systèmes de Détection d'Intrusions, modèle sur lequel ces travaux s'appuient. Puis nous finissons cette section par les différentes méthodes utilisées par ces solutions pour détecter les comportements malicieux, ainsi que les méthodes utilisées pour évaluer ces solutions.

2.1.1 Définitions

Nous définissons ici les notions importantes à la compréhension de cet état de l'art. Tout d'abord, nous étudions la sécurité réseau. Pour bien cerner les enjeux de la sécurité réseau, nous nous intéressons ensuite aux attaques réseaux et aux différentes attaques considérées dans ce mémoire.

Sécurité réseau

epuis la création des réseaux de machines, la sécurité de ce maillage a toujours été considérée comme fondamentale. Alors que sa sécurité s'intéressait initialement à un nombre de machines peu important, la démocratisation récente d'architectures larges et fortement interconnectées, comme les centres de données ou le Cloud Computing, a changé la donne. En effet, il est maintenant nécessaire de fournir un niveau de sécurité suffisant à un parc de machines grandissant. Il est tout d'abord intéressant de définir ce qu'est la sécurité réseau et les objectifs visés par celle-ci.

Les réseaux sont aujourd'hui une partie essentielle des systèmes d'information. Bishop décrit dans [10] la sécurité des systèmes selon plusieurs critères, tels que : les exigences de sécurité, la politique de sécurité, les mécanismes mis en place et les garanties de cette sécurité. L'ensemble de ces critères définit donc la sécurité d'un système. Nous présentons ces notions dans les prochains paragraphes.

Les exigences de sécurité définissent ce qui doit être sécurisé et dans quelles mesures. Il est évident qu'une PME et une multinationale auront des besoins différents en terme de sécurité. Ces exigences permettent de positionner les besoins d'une entité selon plusieurs critères, tels que l'intégrité, la disponibilité et la confidentialité des données.

La politique de sécurité est l'ensemble de règles qui définit ce qui est permis ou non, selon les exigences de sécurité fixées. L'analyse d'un comportement grâce à cette politique de sécurité permet de fixer si l'acteur est malicieux ou non.

Les mécanismes de sécurité sont les moyens techniques ou théoriques permettant d'appliquer une politique de sécurité. Ils ont la responsabilité de vérifier le comportement des acteurs du système et d'établir si leurs actions sont néfastes pour le système.

Les garanties de sécurité permettent d'établir le degré de confiance qu'un administrateur peut avoir envers ses mécanismes de sécurité ou sa politique de sécurité.

Nous considérons dans ce mémoire la définition de Bishop en ce qui concerne la sécurité réseau. La sécurité réseau est composée de trois éléments principaux : les exigences, la politique et les mécanismes de sécurité. Les exigences fixent les objectifs de la sécurité et répondent à la question : « De quoi doit être capable ma solution de sécurité ? ». La politique de sécurité définit ce qui est permis, selon les objectifs fixés. Elle répond à la question : « Quelles étapes ma solution de sécurité doit suivre pour atteindre les objectifs fixés ? ». Les mécanismes de sécurité sont l'ensemble des outils mis en place pour appliquer la politique de sécurité définie et répondent à la question : « Quels sont les procédures et outils en place pour appliquer les étapes décrites dans la politique de sécurité ? ».

Attaques réseaux

Nous avons défini dans la section précédente la notion de sécurité réseau et ses principaux composants. Dans cette section, nous nous intéressons aux attaques que l'on souhaite détecter dans nos travaux.

Dans [35], Ghorbani *et al.* définissent les attaques réseaux comme l'activité malicieuse visant l'interruption, la dégradation, la perturbation de services accessibles aux travers d'un réseau. L'objectif de ces attaques est de porter atteinte à l'intégrité, la disponibilité ou la confidentialité de ces services. Les attaques sont diverses et variées, et peuvent cibler une machine ou un système complet. Quant à Hansman *et al.*, ils définissent dans [38] les attaques réseaux comme des attaques visant un réseau ou des utilisateurs en manipulant les protocoles réseaux de la couche physique à la couche application. Dans ce mémoire, nous considérons qu'une attaque réseau est une attaque dont le médium de communication est le réseau. Nous présentons dans la suite de cette section des attaques réseaux [1] que nous ciblons dans ces travaux.

Les dépassements de tampon [19, 52] (*buffer overflows* en anglais) sont des opérations

1. Étant donné l'évolution du paysage de la sécurité réseau, cette liste n'est pas exhaustive

qui consistent à prendre le contrôle, à interrompre ou à dégrader un processus. Ces attaques s'appuient sur le concept de l'espace mémoire et de son placement, et écrivent en dehors de son espace mémoire, de manière à effacer des informations importantes pour le processus en question. Bien que ces attaques ciblent un système et non le réseau, leur moyen de propagation privilégié reste le réseau, avec par exemple les attaques XSS (Cross-Site Scripting), qui exécutent des routines lors d'une navigation habituelle sur le web.

Les virus, les vers et les chevaux de Troie [91, 100] sont des programmes dont l'objectif est d'infecter une machine hôte. Les actions alors réalisées par ces programmes sont variées : il peut s'agir d'envoi de courriels indésirables, d'écoute passive ou de recherche d'informations confidentielles, ou encore de manipulation de l'hôte pour réaliser des actions malicieuses. Les vecteurs de propagation de ces attaques sont également très divers, mais la plupart de ces menaces se répliquent en utilisant le réseau. Qu'il s'agisse d'une pièce jointe dans un courriel, ou bien de recherche active de victimes potentielles, la phase de réplication peut être détectée en cherchant des motifs particuliers dans les communications portées par le réseau.

Les botnets [29] sont des réseaux de machines composés de dizaines jusqu'à des milliers de machines infectées, utilisées à l'insu de leur propriétaire pour perpétrer des attaques réseaux. L'hôte infecté, aussi appelé *zombi* ou *bot*, peut être infecté grâce aux attaques décrites dans les paragraphes précédents. Le serveur de contrôle peut alors dicter les actions que les bots doivent réaliser, à distance et au travers d'un canal de communication spécifique et sécurisé. Une fois formé, un botnet peut être monétisé : vous louez à la durée ou au service un certain nombre de machines infectées.

Le Déni de service [102, 65] (*Denial of Service* en anglais) est une attaque dont l'objectif principal est de dégrader les performances d'un système. Ces attaques peuvent cibler différents composants du système, tels que la mémoire, le processeur ou encore la carte réseau. La popularité de ces attaques est grandissante ces dernières années, à tel point que l'on parle maintenant de *Distributed Denial of Service* (DDoS) où l'attaque est menée non plus par un seul système, mais par un grand nombre d'hôtes attaquants. Le vecteur de propagation par excellence de cette attaque est le réseau.

Les attaques coordonnées [105], également appelées attaques distribuées, consistent à morceler une attaque et la réaliser à partir de plusieurs hôtes. L'objectif de cette distribution est de pouvoir échapper aux solutions de sécurité, qui attribuent généralement la source d'une attaque à une seule machine. Zhou et al. [105] catégorisent ces attaques suivant le nombre de machines impliquées, qu'il s'agisse des sources ou des cibles de l'attaque distribuée. Ils parlent notamment d'attaques *one to many* lorsqu'une attaque provient d'une seule machine, mais qui est dirigée vers un grand nombre de cibles. À l'inverse, des attaques *many to one* proviennent de plusieurs machines et ciblent une seule machine. Ils soulignent également que ces attaques distribuées sont appliquées sur des réseaux de machines de type *botnet*, que nous présentons dans le prochain paragraphe.

Pour finir, les **attaques réseaux** regroupent quant à elles toutes les attaques qui s'appuient

sur les différents protocoles réseaux. En plus des attaques déjà présentées, nous pouvons évoquer les attaques qui utilisent des biais des protocoles (anomalies ou comportements non fixés), l'usurpation d'identité, l'interception de messages réseaux ou encore les analyses du trafic réseau qui visent à déterminer des informations sur les systèmes à cibler [90]. Nous proposons dans nos travaux un moyen de décrire toutes les attaques présentées dans cette section.

2.1.2 Solutions de sécurité existantes

Nous avons défini dans la section précédente les différents composants de la sécurité réseau. Dans celle-ci, nous présentons particulièrement les mécanismes de la sécurité, que nous appelons aussi *solutions de sécurité*. Ces solutions de sécurité sont les outils mis en place pour appliquer la politique de sécurité, selon les exigences de sécurité fixées. Les trois solutions de sécurité principales (pour sécuriser le réseau) sont les pare-feux, les systèmes de détection d'intrusions et les systèmes de prévention d'intrusions. Ces différentes solutions sont présentées ci-dessous et nous discutons de leurs avantages et inconvénients.

Motivations

Initialement, le réseau mondial *Internet* était utilisé par une petite communauté d'utilisateurs partageant les mêmes valeurs : ouverture, partage et collaboration. Cette approche du réseau a été bouleversée à la fin des années 80, à l'apparition de plusieurs vers dont le ver Morris [87]. Dès lors, il fut clair que internet était également composé de personnes malicieuses.

Puisqu'il n'est pas possible de faire confiance à l'ensemble des utilisateurs, des mécanismes ont été mis en place pour assurer un certain niveau de confiance dans le réseau et ses utilisateurs. Ces mécanismes ont pour objectif de vérifier des propriétés de bases que sont :

— **la disponibilité**, qui assure que les services du système sont accessibles à tout moment, dans un délai raisonnable ;

— **la confidentialité**, qui assure que les informations confidentielles des utilisateurs restent secrètes. Autrement dit, cela consiste à rendre l'information inintelligible à d'autres personnes que son propriétaire ;

— **l'intégrité**, qui assure que les données ne sont pas modifiées ou effacées par des utilisateurs non autorisés. Le système doit être capable de vérifier que les données n'ont pas été altérées durant une communication ;

— **l'authentification**, qui assure que lorsqu'un utilisateur agit sur les services mis en place, le système est capable de garantir que l'utilisateur est bien celui qu'il prétend être ;

— **la non-répudiation**, qui assure que lorsqu'un utilisateur agit sur les services, il ne lui est pas possible de nier d'avoir fait cette action.

Firewalls

Selon Bellovin *et al.* [8], un pare-feu (ou firewall en anglais) est une collection de composants placée entre deux réseaux. Ils ajoutent qu'un firewall doit filtrer l'ensemble du trafic qui provient du réseau externe dirigé vers le réseau interne (et vice-versa). De même, un firewall filtrera le trafic non autorisé, selon la politique de sécurité fixée. Pour finir, par définition, un firewall devrait être impénétrable et inattaquable.

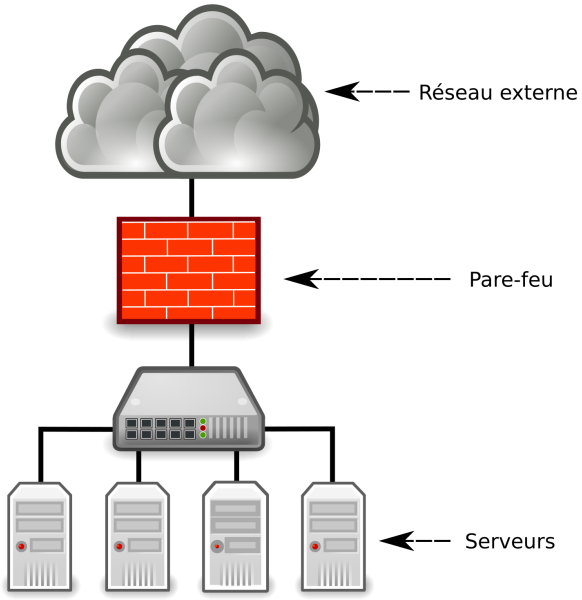


FIGURE 2.1 – Placement d'un firewall dans une architecture réseau

La Figure 2.1 représente le placement d'un firewall dans une topologie réseau. Cet équipement est le nœud qui fait l'intermédiaire entre le réseau interne et le réseau externe. Cette structure propose un désavantage certain, comme le font remarquer Bellovin *et al.* dans [9, 44] : il est nécessaire de faire confiance à l'ensemble des hôtes connectés dans le réseau interne.

Les auteurs soulignent également que la quantité d'information échangée sur le réseau ne cesse de croitre, tant par le nombre d'hôtes communiquant que par la quantité de données à envoyer par message pour certains protocoles. C'est pourquoi ces firewalls ont tendance à devenir des points de congestion. D'autre part, dans de tels réseaux, les points d'entrée d'un réseau sont voués à être multipliés, pour différentes raisons : pour des soucis de performance, de distribution de charge, etc. Il est donc difficile d'appliquer une politique de sécurité cohérente pour un trafic réseau qui peut traverser plusieurs points d'entrée différents sans mécanisme de corrélation ou de communication entre ces nœuds.

Malgré ces désavantages, les firewalls sont utilisés dans les architectures réseaux courantes [60]. Ils font généralement office de solution de sécurité de première ligne. Toutefois, ils assurent un grand nombre de fonctionnalités réseau annexes, telles que la translation d'adresse réseau (NAT) ou les proxy web. L'expertise de ces systèmes varient suivant les produits. Nous avons catégorisé ces niveaux d'expertise dans le Tableau 2.1.

Type de firewall	Caractéristiques
Filtrage statique	— Autorise/refuse les paquets réseaux en inspectant seulement les informations de l'entête, comme par exemple la source ou la destination du paquet, les ports, etc — Ne peut pas détecter le code malicieux dans le contenu du paquet, ni le spoofing ou les attaques par fragmentation
Filtrage à état	— Conserve en mémoire des informations supplémentaires, par exemple les connexions TCP actives — Pour les protocoles tels que TCP, filtre les paquets qui ne se rapportent pas à une connexion existante — Nécessite davantage de mémoire pour maintenir en mémoire ces informations complémentaires
Firewall applicatif	— Évolution du filtrage à état — Analyse le contenu de la couche application — Inspecte le contenu des paquets pour détecter du code malicieux — Capable d'approfondir son analyse en fonction des protocoles, par exemple FTP qui utilise plusieurs ports pour échanger les données ou les commandes du protocole — Nécessite davantage de ressources pour conduire une analyse

TABLE 2.1 – Tableau récapitulatif des différents types de firewalls

Les firewalls sont des composants puissants de la sécurité. En effet, ils constituent une bonne défense contre les attaques basées sur les couches basses du modèle OSI : détection de spoofing IP, de fragmentation de paquets TCP, etc. Toutefois, il y a un certain nombre de comportements malicieux qu'ils ne peuvent détecter, notamment sur les couches hautes du modèle OSI [42, 60]. Ils ne sont par exemple pas capables d'analyser le contenu des échanges transitant par des VPN. De plus, par leur placement habituel en bordure du réseau, ils ne peuvent pas détecter le comportement malicieux provenant de l'intérieur même du réseau qu'ils doivent protéger.

Dès 1999, plusieurs articles [9, 44] s'orientent vers des solutions distribuées pour cibler ces attaques internes. Les premières implémentations visent à répliquer le filtrage à plusieurs endroits du réseau de manière autonome, sans prise de décisions communes. D'autres travaux sur la distributions de solutions de sécurité sont présentés dans la Section 2.1.3, qui traite des systèmes de détection d'intrusions.

Systèmes de détection et de prévention d'intrusions

Nous avons discuté dans la section précédente des firewalls, les solutions de sécurité de première ligne, qui ont comme désavantage leur localisation en bordure du réseau. Contrairement aux firewalls, les Systèmes de Détection d'Intrusions (IDS) et les Systèmes de Prévention d'Intrusions (IPS) sont placés arbitrairement dans le réseau à surveiller. Nous présentons brièvement dans cette section ces deux types de solutions de sécurité.

Debar *et al.* définissent dans [25] les IDS/IPS comme des systèmes capables de surveiller en temps réel un système et de détecter qu'une séquence d'actions est symptomatique d'un comportement anormal ou malicieux. De manière plus macroscopique, ces systèmes analysent les événements du système à protéger et détectent les attaques ou comportements illégitimes, à l'instar des firewalls. La différence fondamentale entre les IDS et IPS est leur comportement après avoir statué qu'une attaque était en cours. Les IDS se satisfont d'un système d'alarmes, qui a pour but de prévenir le responsable qu'une attaque est en cours. Leur rôle est alors de proposer une vue synthétique de la sécurité du système et d'offrir des informations d'aide à la décision, pour que les responsables puissent agir en connaissance de cause. Quant aux IPS, ils mettent en place des contre-mesures et tentent ainsi de mettre fin aux attaques dans un délai raisonnable.

Ces systèmes se basent sur trois types d'information : le contexte du système (un ensemble d'informations sur le long terme lié aux méthodes de détection utilisées), la configuration du système et les événements à surveiller.

Dans le contexte d'un réseau composé de nombreux systèmes informatique, une vue locale des événements est généralement insuffisante pour détecter une menace et il est nécessaire de se concerter avant de déduire qu'une attaque est en cours. C'est pourquoi nous focaliserons nos travaux sur les IDS, qui nous permettront de corréler les informations au travers des alertes générées. Ces IDS sont présentés plus en détails dans la Section 2.1.3

2.1.3 Systèmes de détection d'intrusions

Dans la section précédente, nous avons présenté brièvement les Systèmes de Détection d'Intrusions (IDS en anglais). Ces équipements sont définis dans [4] comme des systèmes d'alarmes dans le domaine de la sécurité informatique. Ces systèmes ont comme rôle d'alerter les entités (généralement le responsable de la sécurité des systèmes) lorsque des comportements anormaux ou malicieux sont détectés. C'est alors à ces entités de prendre en charge la mise en place des contre-mesures.

Dans cette section, nous proposons une présentation de ces IDS, basée sur une étude de plusieurs taxonomies majeures dans la littérature [25, 4, 60]. Bien que ces travaux divergent dans les classifications qu'ils proposent, ils s'accordent sur un certain nombre de catégories principales. La Figure 2.2 propose une classification simplifiée de ces systèmes. Nous détaillons l'ensemble de

ces points dans les paragraphes suivants.

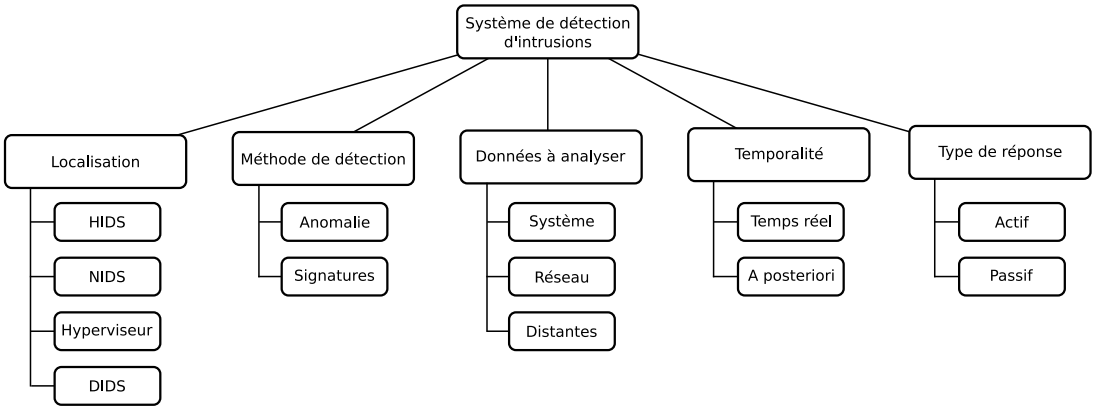


FIGURE 2.2 – Classification des IDS

Localisation du Système de Détection d'Intrusions

La première caractéristique de ces IDS est leur emplacement dans la structure à surveiller. Qu'ils s'agissent d'IDS placés en coupure sur le réseau ou directement sur chaque machine de l'infrastructure, le placement de ces systèmes implique un certain nombre de conséquences, telles que la capacité ou non de détecter certains types d'attaques, la responsabilité du déploiement ou de mise à jour de ces systèmes, etc. Nous détaillons dans les paragraphes suivants chacun des placements possibles.

Les **Host-based IDS** (HIDS) [45, 84] sont placés directement sur les systèmes hôtes à surveiller. Ils analysent les fichiers, appels système ou événements réseau de la machine hôte. Ils sont par conséquent installés par l'administrateur du parc de machines ou directement par l'utilisateur. Également, la détection d'intrusions est limitée au poste en question.

Les **Network-based IDS** (NIDS) [64, 76] sont placés sur le réseau, à proximité des équipements réseau. Généralement en coupure du réseau, ils peuvent être directement intégrés dans les routeurs. Ils détectent les intrusions grâce à l'analyse du trafic réseau et sont capables de surveiller plusieurs systèmes.

Les **Hypervisor-based IPS** [34] sont placés sur un système qui propose des mécanismes d'hypervision. Ces systèmes peuvent héberger plusieurs machines virtuelles (VM), qui s'exécutent sur le même hôte physique. Le rôle de ces IDS est d'analyser le trafic réseau entre les VM et entre l'hyperviseur et les VM. Ces systèmes restent complexes à mettre en place, à cause de leur criticité et la difficulté à s'interfacer avec l'hyperviseur.

Les **Distributed IDS** (DIDS) [85, 66, 101, 92] sont des systèmes composés de plusieurs IDS, capables de se coordonner dans la détection des intrusions. Ils se reposent sur de multiples IDS, disséminés sur les machines hôtes du parc à surveiller ou directement sur le réseau. Les DIDS

restent des systèmes très complexes, notamment à cause des problématiques de coopération, de communications riches entre les composants du DIDS ou de la distribution des données. Nous présentons plus en détails les DIDS dans la Section 2.1.4.

Modi *et al.* proposent dans [60] un tableau récapitulatif, mettant en avant les avantages, désavantages de chacun de ces IDS. Ils discutent également dans ce tableau de la responsabilité du déploiement et de mise à jour de ces systèmes.

Type d'IDS	Avantages	Désavantages	Placement	Déploiement & responsabilité
HIDS	— Détecte les intrusions en surveillant les fichiers, appels système ou événements réseau de l'hôte — Pas besoin d'équipement en plus	— Besoin de l'installer sur chaque machine — Détection d'attaques locales uniquement	Machine virtuelle ou physique	Utilisateur & administrateur
NIDS	— Détecte les intrusions en surveillant le trafic réseau — Besoin d'être placé sur le réseau (physiquement) — Peut surveiller plusieurs systèmes en même temps	— Difficile de détecter des intrusions provenant de contenu chiffré — Ne peut pas détecter les attaques ne transitant pas par le NIDS	Réseau physique ou virtuel	Administrateur
Hypervisor-IDS	— Détecte les intrusions entre les VM en analysant le trafic réseau	— Récent et difficile de s'interfacer avec les hyperviseurs — Composant critique	Hyperviseur	Administrateur
DIDS	— Caractéristiques des HIDS/-NIDS — Détecte les intrusions en associant plusieurs systèmes de détection d'intrusions (HIDS/NIDS)	— Coût de déploiement et de configuration — Surcoût en communication — Coopération entre les systèmes complexe	Partout	Utilisateur & Administrateur

TABLE 2.2 – Tableau récapitulatif des différents placements des IDS

Méthodes de détection

Ces systèmes se différencient également en matière de méthodes utilisées pour détecter les intrusions. Initialement, il existait deux grandes familles de techniques de détection, mais avec l'arrivée des IDS distribués (DIDS), de nombreuses techniques ont vu le jour. Nous détaillons dans cette partie les méthodes de détection utilisées par ces systèmes.

La **détection d'anomalies** est une technique de détection qui n'est pas axée sur la recherche d'intrusions. Elle se concentre sur l'analyse d'un comportement en le comparant à un modèle considéré comme normal. Axelsson *et al.* définissent dans [4] que cette technique consiste à considérer comme malicieux tout comportement qui s'écarte de la normalité. Le comportement « normal » est généralement extrait de documents tels que des spécifications de standards (RFC du protocole de TCP [68] par exemple). Dans [25], Debar *et al.* ajoutent que ce comportement légitime peut être appréhendé lors d'une phase d'apprentissage, où le système de détection d'intrusions apprend en observant un hôte sain.

Cette méthode se base sur plusieurs principes pour détecter les intrusions : systèmes experts, statistiques ou encore apprentissage. Le point commun entre toutes ces méthodes est qu'il s'agit

ici de mettre en évidence qu'un comportement est suspicieux. La détection d'anomalies a pour principal avantage qu'elle permet de détecter de nouvelles attaques sans qu'il y ait besoin de redéfinir la politique de sécurité. Par contre, cette technique peut se révéler assez peu précise et génère un grand nombre de faux-positifs. Également, les modèles peuvent se révéler incomplets ou sujets à des utilisations abusives. Dans ce cas, cette méthode est incapable de détecter les attaques.

La **détection par signatures** est le pendant de la méthode précédente. En effet, alors que la détection d'anomalies se concentre sur la définition d'un comportement légitime, la détection par signatures se penche plutôt vers la modélisation des comportements connus comme étant malveillants [25, 4, 60]. Cette technique de détection est basée sur la description de ces comportements suspects au travers de règles de description, appelées signatures. Ces signatures peuvent alors décrire des machines à états, des recherches de motifs ou encore des analyses statistiques. Le principal désavantage de cette technique réside en son besoin d'avoir une base de règles mise à jour régulièrement. En effet, le système de sécurité sera incapable de détecter de nouvelles attaques, puisqu'il est nécessaire que la règle associée soit décrite dans la base de règles. Toutefois, cette méthode est fiable car elle génère peu de faux-positifs.

D'autres méthodes de détection existent, mais que nous n'aborderons pas dans les détails dans ce mémoire. Parmi elles, nous pouvons citer les réseaux de neurones, les algorithmes génétiques ou encore les méthodes d'association, qui permettent de détecter des variations d'une attaque. De même, des méthodes hybrides existent, qui composent plusieurs des techniques présentées dans cette section. Pour finir, depuis l'apparition des IDS distribués, il est nécessaire de pouvoir corréler des événements qui se produisent en de multiples localisations.

Type de réponse en cas de détection d'intrusions

Les différentes taxonomies [25, 4, 60] s'accordent sur cette catégorie. Il existe deux types de réaction lors d'une détection d'intrusions : la passivité ou l'activité. Les IDS **passifs** se satisfont de notifier l'autorité de sécurité qu'une attaque est en cours et ne tentent pas d'arrêter ou de mitiger l'intrusion. Les IDS **actifs** cherchent quant à eux à arrêter l'attaque en cours et à isoler l'attaquant. Cette opération peut consister à modifier l'état d'alerte de l'IDS, à fermer des connexions réseaux, en tuant des processus malfaiteurs, etc.

Dans nos travaux, nous ne considérons que des systèmes de détection d'intrusions passifs, qui alertent l'autorité en charge de la sécurité du réseau. En effet, la structure réseau pouvant être complexe, la question de savoir comment agir pour mitiger l'attaque en cours est une problématique en soit, aussi complexe que celle de la détection.

Source des données à analyser

Les systèmes de détection d'intrusions conduisent leurs analyses sur des données qui proviennent de diverses sources. Nous pouvons établir deux catégories principales : les données issues du trafic réseau et les données issues des systèmes (logs du noyau, appels système, alertes de sécurité, etc.) [4]. La source des données à analyser est à mettre en parallèle avec la localisation de l'IDS lui-même : un HIDS se focalisera davantage sur l'analyse de données provenant du système alors qu'un NIDS se concentrera sur des données réseau.

Avec l'apparition des IDS distribués, une troisième catégorie a émergé. En effet, les données à analyser ne sont plus directement les données brutes relevées par le système de sécurité lui-même, mais des données récoltées par d'autres sondes. Ces sondes rapatrient leurs alertes vers des nœuds qui seront alors responsables de l'agrégation de l'ensemble des alertes [105].

Temporalité de la détection

Les systèmes de détection d'intrusions analysent des données pour établir si une attaque est en cours. Cette analyse peut être en temps réel ou bien réalisée après capture des événements à étudier. Nous parlons donc de système **temps réel** quand le système parvient à traiter les événements *à la volée*, et de système **a posteriori** lorsque le système réalise ses analyses *post mortem* alors que tous les événements qui ont eu lieu son disponibles. Cela a une forte incidence sur le choix des algorithmes. Par exemple, les algorithmes polynomiaux sont souvent rédhibitoires dans le premier cas, contrairement au second.

Dans nos travaux, nous considérons uniquement des IDS qui conduisent leurs analyses en temps réel, puisque nous désirons que les alertes soient acheminées le plus rapidement possible aux autorités responsables de la prise de décisions.

Systèmes de détection d'intrusions existants

Dans cette section, nous présentons plusieurs IDS existants dans la littérature. La liste des solutions présentées n'est pas exhaustive. Elle est cependant composée de solutions représentatives de l'ensemble des catégories décrites dans les paragraphes précédents. Certaines solutions, telles que Snort [76] ou Bro [64], sont également très populaires et la communauté utilise ces solutions comme des éléments de comparaison. En ce qui concerne les solutions distribuées et collaboratives, nous présentons des solutions existantes dans la Section 2.1.4.

Haystack est un prototype proposé par Smaha [84] en 1988, afin de détecter les intrusions sur des systèmes multi-utilisateurs de l'Air Force. Pour cela, ce système utilise à la fois de la détection d'anomalies et également de la détection par signatures. Cet HIDS est organisé suivant deux concepts majeurs : un historique du comportement des utilisateurs et un modèle qui définit

comment les utilisateurs peuvent se comporter.

IDES [54] est un projet qui propose un système de détection d'intrusions assez classique. Les travaux autour de IDES ont été complétés au travers de NIDES [2]. L'hypothèse initiale adoptée par IDES est que chaque utilisateur se comporte de manière cohérente dans le temps et qu'il est alors possible de se servir d'outils pour modéliser leur comportement, tels que les statistiques. Il est alors possible de comparer les événements aux modèles et d'en déterminer si une intrusion est en cours. NIDES est une extension de ce travail et utilise plusieurs sondes IDES qui lui transmettent les événements locaux.

NSM [39, 61] est le premier prototype à considérer le trafic réseau comme objet d'étude principal. NSM écoute de manière passive le trafic réseau et détecte les intrusions en se basant sur un système expert qui recherche des motifs malicieux. Souhaitant supporter des systèmes hétérogènes, ils se sont orientés vers l'analyse du trafic réseau qui est standardisé. Placé en coupure sur le réseau à surveiller, NSM utilise également des outils statistiques pour détecter les attaques.

Snort [76] est un NIDS qui utilise principalement deux méthodes de détection : la détection par signature et la détection d'anomalies. Cette solution était supportée durant plus d'une décennie par Sourcefire et a été récemment acquise par Cisco. Elle se repose sur une communauté qui enrichit la base de signatures et utilise également des modules spécialisés pour certains types d'intrusion. La communauté de Snort et sa popularité dans la littérature fait d'elle une solution de choix comme élément de comparaison.

Bro [64] est apparu au même moment que Snort, avec les mêmes intentions : offrir un moyen de décrire des signatures pour détecter le trafic malicieux et également se baser sur des modules spécialisés. Pour décrire ces signatures, Bro propose un langage dédié qui se rapproche d'un langage impératif classique et permet de décrire les motifs des attaques. Sa communauté est moins importante que celle de Snort, mais il faut remarquer que Bro est la première solution à considérer que le système d'analyse puisse être la cible d'une attaque. Dans l'article, Paxson [64] distingue plusieurs catégories d'attaque :

— les attaques par déni de services, qui visent à rendre le système incapable de continuer son analyse ;

— les attaques qui tentent de faire planter la solution de sécurité ;

— les attaques dont l'objectif est d'utiliser des subterfuges pour biaiser l'analyse de la solution.

Bro propose des mécanismes pour contrer ces différents types d'attaques.

Terra [34, 33] est un IDS placé au niveau de l'hyperviseur. Son placement le rend différent d'un HIDS, qui analyse les événements d'un système. Proposé pour les architectures qui utilisent les mécanismes de virtualisation, Terra peut analyser les communications (entre les machines virtuelles ou entre une machine virtuelle et l'hyperviseur) et peut également accéder à la mémoire des VM. Pour conduire ses analyses, Terra se base sur la détection d'anomalies. Il communique avec le composant VMM (*Virtual Machine Monitor*), responsable de la virtualisation matérielle, de l'isolation mémoire et de la surveillance des machines invitées, pour obtenir les données à étudier.

2.1.4 Systèmes de détection d'intrusions distribués et collaboratifs

Nous avons vu dans la section précédente que les systèmes de détection d'intrusions se déclinaient sous différentes formes, suivant leur placement et le système surveillé. Nous avons mis en avant l'existence des IDS distribués, qui correspondent à une composition de plusieurs IDS, potentiellement hétérogènes. Vasilomanolakis *et al.* définissent dans [96] les IDS distribués et collaboratifs comme des systèmes composés de sondes, responsables de la collecte, et de nœuds d'analyse, qui se chargent de la détection d'intrusions. Ce type de structure permet d'analyser plus précisément et avec plus de fiabilité les événements des systèmes à surveiller. Les deux éléments principaux de cette structure sont les collecteurs et les analyseurs. Initialement, ces deux types de composants étaient clairement distingués, mais les dernières propositions de la littérature s'orientent maintenant vers des sondes capables à la fois de collecter et d'analyser les données. Nous discutons dans cette section de ces IDS collaboratifs (CIDS), en décrivant les structures de ces systèmes et leurs méthodes de détection.

Motivations

Cherchant initialement la notoriété, les pirates informatiques s'orientent maintenant vers le profit. Leur but est de s'enrichir en utilisant les failles des systèmes ciblés. Qu'il s'agisse de spam (envoi massif de mails non sollicités), de phishing (vol d'identité sur internet) ou encore de dégradation de services grâce à des attaques de déni de services, ces pirates ne visent plus uniquement la popularité [30]. Pour accélérer cet enrichissement, ils utilisent maintenant des attaques à grande échelle, telles que les scans de ports distribués [90, 91], les vers [100] ou les attaques par déni de services distribuées [30, 65]. Ces attaques peuvent être automatisées et lancées à partir de multiples réseaux, ce qui rend difficile la tâche de détection d'intrusions. Effectivement, le morcellement des attaques permet de rendre moins suspect un comportement malveillant [105]. C'est pour réussir à détecter ce type d'attaques que les systèmes de détection d'intrusions collaboratifs (CIDS) sont nés. Ces systèmes sont capables d'analyser des événements provenant de multiples réseaux et de les corréler afin de déduire qu'une attaque à grande échelle est en train de se produire.

Architecture des IDS collaboratifs

Les CIDS sont des systèmes composés de plusieurs IDS, hétérogènes ou non. Les différentes solutions proposées dans la littérature se basent sur plusieurs approches. Nous les présentons en détail dans les prochains paragraphes.

Approche centralisée La première approche qui apparait dans la littérature est l'approche centralisée, qui est composée de deux types de nœuds : les nœuds collecteurs et les nœuds d'analyse. Les nœuds collecteurs sont des IDS qui opèrent de manière locale et reportent les intrusions au nœud central, qui corrèle les informations.

La Figure 2.3 illustre ce type de structure. Dans cette figure, nous avons schématisé l'approche présentée, qui repose sur deux types de composants. Les nœuds de détection sont placés sur le réseau et analysent localement les données. Lorsqu'une intrusion est détectée, ils envoient des informations au nœud central, qui va pouvoir corréler les données et détecter si un motif global d'attaque est en cours.

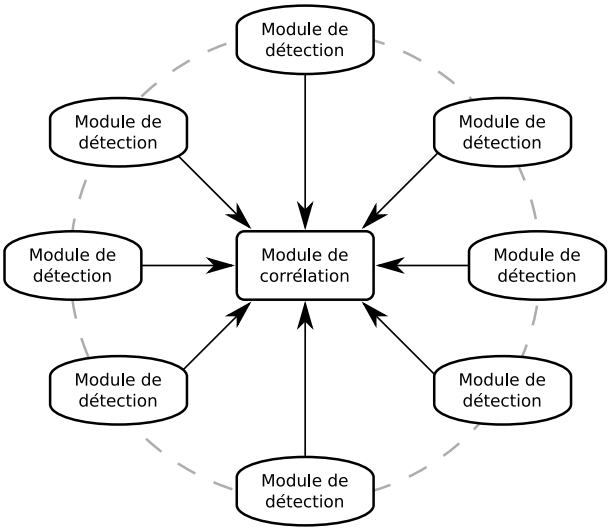


FIGURE 2.3 – Approche centralisée

DIDS [85] est un CIDS qui se base sur cette approche. Proposée par Snapp *et al.*, Cette solution de sécurité est composée de trois types de composants : un système expert (le nœud central), des HIDS et des NIDS. Les IDS collectent les données locales et les analysent. Une fois qu'une donnée malveillante est détectée, elle est envoyée au système expert, grâce à un format de message homogène. Le système expert décide ensuite s'il y a attaque globale ou non. Selon Vasilomanolakis et al. [96], DIDS se repose sur une méthode de détection simpliste qui peut facilement être outrepassée. De plus, ils ajoutent que le surcoût (en terme de communication ou de calcul) augmente considérablement en fonction de la taille du réseau surveillé. Pour finir, le système expert est un point individuel de défaillance (*single-point of failure* en anglais, ou

SPOF), ce qui remet en cause la résistance aux fautes de cette solution. NetSTAT [97] est une solution assez similaire à DIDS. Elle se base également sur une architecture client-serveur. Les IDS collectent les données locales et envoient les alertes à un serveur central. Le serveur central détecte les intrusions en utilisant un mécanisme de machine à états, décrit grâce à des signatures. Quant à la solution CRIM [20], l'architecture est similaire. Leur méthode de détection se base sur un algorithme qui recherche les similarités entre les alertes envoyées au serveur central.

Pour conclure sur cette approche, les CIDS centralisés ont généralement un très bon taux de détection d'intrusions. Mais il y a deux désavantages majeurs pour cette approche [105]. Le premier concerne le nœud central, qui est un point individuel de défaillance. Si le nœud d'analyse est désactivé, la corrélation d'alertes n'est plus utilisable. D'autre part, en cas d'attaque coordonnée, le noeud d'analyse devra être capable de gérer l'ensemble des alertes envoyées par les noeuds collecteurs. Pour de grosses attaques, il est possible que des alertes soient ignorées ou que la détection soit différée.

Approche hiérarchique Pour éviter que le nœud central soit un SPOF, l'approche hiérarchique propose que plusieurs nœuds soient responsables de ces opérations de corrélation. Le système est décomposé en plusieurs groupes de communication. Chaque groupe est un sous ensemble de la hiérarchie fixée. Un IDS est désigné comme noeud d'analyse par groupe et est responsable de la communication avec les groupes de niveau supérieur. Cette approche a été appliquée par de nombreuses solutions, dont GrIDS, AAFID ou encore EMERALD [66].

La Figure 2.4 représente cette approche. Nous illustrons dans cette figure trois groupes de communications, composés chacun de modules de détection ou de corrélation. Pour le groupe 1, un noeud est désigné pour réaliser les opérations de corrélation, ainsi que la partie de communication avec la couche supérieure. Le groupe 3 est alors capable de corréler des informations venant à la fois du groupe 1 et du groupe 2.

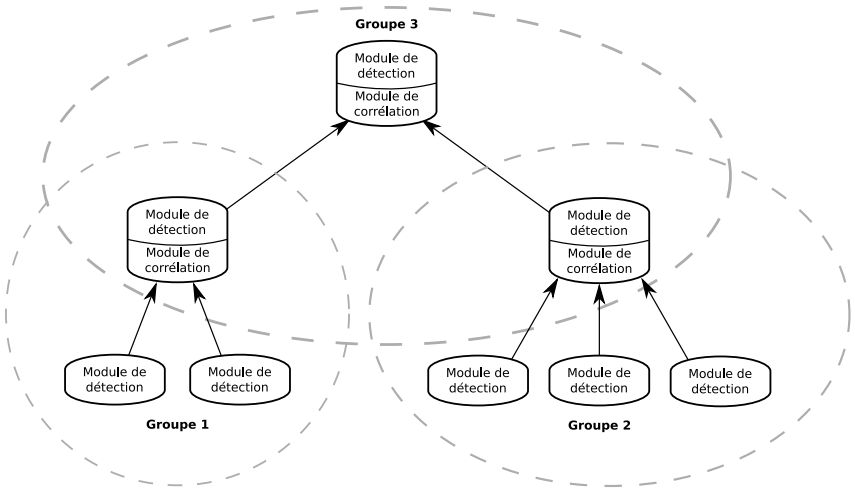


FIGURE 2.4 – Approche hiérarchique

Basée sur cette approche, *Graph Based Intrusion Detection System* (GrIDS) [92, 15] est une solution destinée à être déployée sur de larges réseaux. Le réseau de protection est séparé en *départements*, qui sont eux mêmes organisés grâce à une structure en arbre. Chaque département contient un système d'analyse et de multiples NIDS et/ou HIDS. Chacun de ces IDS analyse les événements locaux et envoie les alertes au système d'analyse du département. D'autre part, chaque département contient un moteur de graphe et un module de communication. Ce dernier se charge de la communication entre les départements. Le moteur de graphe est responsable de l'analyse des événements locaux et de l'agrégation de ces informations, qui seront ensuite transmises au niveau supérieur si nécessaire. *Autonomous Agents For Intrusion Detection* (AAFID) [6, 88] est également basé sur cette approche hiérarchique. Contrairement à la solution précédente, celle-ci n'est pas décomposée en départements, mais les différents rôles des sondes en jeu sont les mêmes. AAFID est composé de trois types de sondes : des agents, des récepteurs et des moniteurs. Les agents analysent les événements locaux et génèrent des rapports d'alerte lorsqu'une intrusion est détectée. Ces rapports sont envoyés aux récepteurs, qui supervisent les agents et corrèlent les informations. Les moniteurs reçoivent des rapports synthétiques de la part des récepteurs et peuvent corréler les rapports de plusieurs récepteurs. Quant à *Event Monitoring Enabling Responses to Anomalous Disturbances* (EMERALD) [66], il s'agit d'un CIDS qui définit plusieurs domaines de surveillance : service, département et entreprise. Cette notion de domaine s'approche de la notion de département de GrIDS, mais EMERALD permet d'établir des départements de départements. Chaque domaine est responsable de la détection d'intrusions dans son groupe et envoie des rapports à la couche supérieure, qui sera capable de corréler des informations venant de plusieurs domaines inférieurs.

L'approche hiérarchique passe plus facilement à l'échelle que l'approche centralisée. Cependant, si le nœud responsable de la communication entre deux niveaux de groupes vient à être désactivé, une branche entière de la structure est déconnectée de la hiérarchie. Pour finir, les nœuds de niveau élevé sont généralement peu précis en ce qui concerne la correlation d'alertes, car les données envoyées sont généralement incomplètes ou trop abstraites.

Approche distribuée Le désavantage commun des deux approches précédentes est l'existence d'un ou plusieurs points de défaillance. Si ces points viennent à être désactivés, la corrélation ne peut plus avoir lieu. C'est pourquoi l'approche distribuée a été proposée. Celle-ci se repose sur une structure entièrement composée de nœuds à la fois collecteurs et analyseurs. Ces nœuds détectent localement les attaques et sont capables de corréler les informations des nœuds voisins pour détecter les attaques coordonnées.

La Figure 2.5 illustre cette approche et propose une structure composée de plusieurs nœuds. Chaque nœud est équipé d'un module de détection et d'un module de corrélation. Les nœuds détectent localement les intrusions lorsque cela est possible. Si cela est nécessaire, ils vont faire appel à l'information des nœuds du réseau pour pouvoir compléter l'analyse de leurs données.

Intrusion Detection and Rapid Action (INDRA) [47] est un CIDS distribué, composé de moniteurs qui reposent sur Pastry [77], une table de hachage distribuée (DHT). Tous les moniteurs

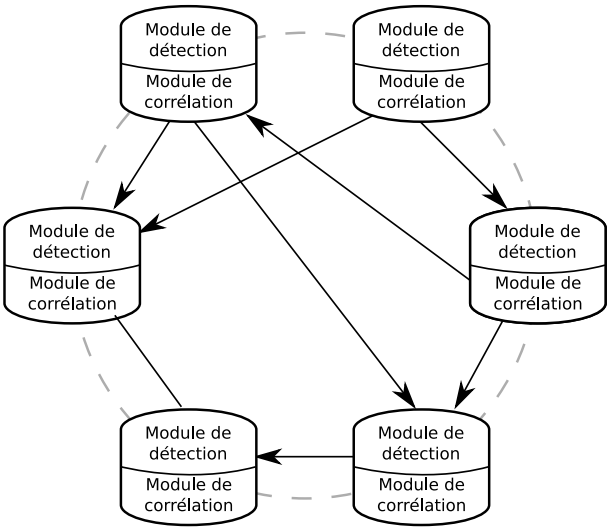


FIGURE 2.5 – Approche distribuée

sont constitués d'un collecteur d'événements et d'un module d'analyse. Lorsqu'une attaque est détectée, un système de défense proactif est activé, pour canaliser l'attaque. Les moniteurs sont capables de corréler les informations du réseau surveillé grâce à la DHT, qui offre des fonctionnalités de recherche de données. DOMINO [101] est une solution basée sur une architecture hybride, avec trois types d'entités : les nœuds *axis*, les communautés satellites et les noeuds contributeurs. Les nœuds *axis* sont les nœuds principaux de cette solution : ils se chargent de la collecte des données et de leur analyse. Ils sont considérés comme fiables et communiquent grâce à un protocole pair à pair. Les communautés satellites sont des réseaux de nœuds satellites, organisées dans un schéma hiérarchique et sont supervisées par un nœud *axis*. Les nœuds contributeurs sont d'autres solutions de sécurité, jugées non fiables, qui peuvent contribuer à la sécurité du réseau en sa globalité.

L'avantage de l'approche distribuée est sa résistance aux fautes : si un noeud tombe en panne ou est infecté, le système sera tout de même capable de continuer sa tâche de corrélation d'alertes. Toutefois, des avis négatifs sont exprimés sur ce type d'approche. Les critiques qui leurs sont faites portent principalement sur deux points : détection d'attaques imprécise ou répartition de la charge inégale.

Méthodes de détection

Nous avons vu dans la Section 2.1.3 les différentes méthodes utilisées pour détecter les intrusions pour une solution de sécurité placée sur le réseau ou une machine hôte. Dans cette section, nous détaillons les méthodes de détection utilisées pour les CIDS. L'ensemble de ces techniques porte sur la recherche de similitudes entre différents événements ; le système cherche à corréler des données qui proviennent de plusieurs points du réseau à protéger.

Corrélation par similitudes Cette approche, appliquée par Valdes and Skinner [94], Debar and Wespi [26] et Cuppens [22], corrèle les alertes générées par les composants de la sécurité en cherchant des similarités entre les données ou les attributs des alertes. Des fonctions permettent de déterminer le score entre deux jeux de données, le score permet alors de déterminer si les données sont corrélées ou non. Cette approche est efficace pour regrouper les événements mais reste généralement peu fiable quand il s'agit d'établir un lien de cause à effet entre deux événements liés et proches dans le temps.

Recherche de scénario d'attaques Les attaques complexes sont généralement réalisées en plusieurs étapes. Par exemple, une phase initiale permet de préparer le système cible, pour qu'il soit plus facilement manipulable. Suite à cette étape initiale, la vraie attaque peut alors avoir lieu [95]. Pour refléter ces attaques complexes, des méthodes de détection établissent des scénarios d'attaques et tentent d'identifier les étapes intermédiaires dans les données à analyser. Cette approche, proposée par Dain and Cunningham [24], Cuppens and Ortalo [23] et Eckmann et al. [27], est très efficace lorsque les scénarios sont détaillés dans la littérature. Cependant, elle est inefficace contre les attaques qui ne sont pas décrites sous forme de scénario.

Corrélation d'attaques multi-étapes La méthode présentée précédemment reste inefficace pour les attaques qui sont inconnues par le système de détection. Pour parer à ce problème, l'approche par corrélation d'attaques multi-étapes a été proposée dans la littérature par Cuppens and Miege [20] et Cheung et al. [16]. Cette approche présume l'existence de relations entre différentes étapes d'une attaque. Pour détecter les attaques, cette méthode fait l'hypothèse qu'une attaque est réalisée pour en préparer une autre. En se basant sur une librairie d'attaques types, le système est capable de détecter les attaques complexes. Cette approche de détection permet de déterminer le lien de cause à effet entre deux étapes d'une attaque, mais le nombre de faux positifs est plus important que dans les approches précédentes. De plus, la librairie d'attaques sur laquelle se repose cette approche doit être assez dense pour que le moteur de détection soit efficace.

Corrélation par filtrage de données Plutôt que de se baser sur la recherche de similarités entre plusieurs alertes, cette approche propose plutôt d'éliminer les alertes non pertinentes. Proposée par Porras et al. [67], Gula [37] et Kruegel and Robertson [51], cette approche se concentre sur les alertes qui auront un fort impact sur le système protégé. Par conséquent, il est nécessaire de spécifier une description du système à surveiller car l'efficacité du moteur de détection repose sur la précision de cette description.

Synthèse Il y a un grand nombre de problèmes soulevés par les méthodes de détection utilisées par les CIDS. Notamment, est-il possible d'avoir une méthode de corrélation expressive et peu coûteuse en calculs ? Ou encore, comment maximiser le taux de détection sans pour autant imposer un surcoût important en communication ou en calculs ? Chacune des approches a des

avantages et des inconvénients et aucune ne parvient à proposer une technique à la fois efficace et expressive. En effet, *la corrélation par similitudes* est efficace mais reste limitée dans sa capacité à découvrir des attaques complexes, décomposées en plusieurs étapes. Quant à la *corrélation par filtrage*, il est nécessaire de donner une description précise de l'ensemble des systèmes à surveiller, ce qui est irréalisable dans un parc de machines important. En ce qui concerne la *recherche de scénarios* ou la *correlation d'attaques multi-étapes*, elles proposent un taux de détection élevé mais nécessitent des analyses coûteuses et des solutions de stockage importantes.

Dissémination des données

Pour pouvoir collaborer et détecter de manière coopérative les attaques, les CIDS doivent échanger des informations quant à leur état courant. Nous discutons dans cette section des méthodes de dissémination des données utilisées par ces systèmes.

La dissémination des données est fortement influencée par l'architecture du CIDS et son mécanisme de gestion des membres du réseau. Les CIDS centralisés proposent une dissémination dirigée exclusivement vers ou depuis le nœud central. En cas d'attaques coordonnées, cela peut provoquer une contention au niveau du nœud central, qui ne serait alors plus capable de gérer l'axe de communication ou d'analyse. Les nœuds d'un CIDS hiérarchique communiquent uniquement avec les nœuds du même groupe ou vers le nœud responsable de la communication avec la couche supérieure.

Les CIDS distribués se reposent sur une architecture qui n'est pas organisée, c'est pour cette raison qu'il faut mettre en place des moyens de communication efficaces. Plusieurs approches ont été proposées dans la littérature. La première approche est basée sur le principe d'inondation sélective : les informations sont envoyées aux nœuds atteignables dans le réseau. Vishnumurthy and Francis [98] proposent d'inonder aléatoirement certains nœuds, alors que Ganesh et al. [31] proposent un protocole pour lequel les composants *murmurent*[2] les informations aux nœuds voisins. La deuxième approche est basée sur le principe du protocole *publish/subscribe* [32]. Les nœuds stockent des informations localement et lorsqu'une modification survient, ils préviennent les nœuds abonnés qu'une modification a eu lieu. La troisième approche se repose sur des réseaux *Peer-to-Peer*[3] (P2P) [53, 101, 103]. Dans cette approche, un nœud du CIDS peut rechercher un contenu sur le réseau P2P établi, le protocole permettant de facilement localiser le nœud qui héberge les données souhaitées. Les tables de hachage distribuées (DHT) [104, 47] sont également des solutions pair à pair utilisées pour distribuer les données entre les nœuds d'un CIDS.

2.1.5 Méthodes d'évaluation des solutions

Nos travaux proposent une nouvelle approche. Pour pouvoir la comparer aux autres solutions existantes de l'état de l'art, nous étudions dans cette section les principales méthodes d'évaluation

2. *Gossiping protocol* en anglo-saxon
3. Pair à pair en français

utilisées [96].

Taux de détection L'objectif principal des IDS est de détecter les intrusions. La première métrique concerne donc le taux effectif de détection. Cette métrique est calculée en faisant un ratio entre le nombre d'attaques effectivement détectées et le nombre d'attaques réalisées. Elle peut être complétée par l'analyse du nombre de faux-positifs, qui met en avant les événements considérés malveillants à tort. Cette métrique est toutefois subjective, car le taux de détection dépend du jeu d'attaque utilisé et de la configuration des machines impliquées dans les simulations.

Rapidité de traitement Pour les solutions qui analysent les données en temps réel, il est nécessaire de proposer un moyen de supporter un débit élevé. Plus la solution est capable d'analyser des paquets réseaux à la seconde, plus le débit supporté est élevé. Plusieurs métriques [64] peuvent être considérées, telles que la mesure du débit maximal supporté sans perte, ou bien la durée moyenne de traitement pour un paquet.

Occupation mémoire Les solutions de sécurité peuvent être localisées à de multiples endroits sur le réseau. Lorsqu'il s'agit d'une solution intégrée sur un serveur à surveiller (un HIDS), il est nécessaire que l'empreinte mémoire soit la plus faible possible, pour éviter de consommer les ressources dédiées au serveur ou pour éviter de créer un déni de services. Plus cette empreinte est faible, moins la solution nuira au bon fonctionnement du système hôte.

2.2 L'émergence du Cloud Computing et son impact

Dans cette section, nous présentons le *Cloud Computing*, une architecture complexe composée de machines et d'équipements réseaux, qui permet d'offrir aux clients des ressources adaptées à leurs besoins. Après avoir défini ce que comporte le Cloud Computing, nous nous concentrons sur l'aspect sécurité d'une large structure réseau telle que le Cloud.

2.2.1 Cloud Computing : une infrastructure complexe

Le *Cloud Computing* est une architecture complexe, en vogue depuis un peu moins de dix ans. Cette architecture a pour but de proposer des services aux clients, qui sont proportionnés selon leurs besoins. Dans cette partie, nous donnons une définition au terme *Cloud Computing* et nous décrivons les grandes notions du Cloud.

Définition du Cloud Computing

Bien que le Cloud Computing connaisse une popularité grandissante depuis l'année 2008, la notion d'architecture qui met à disposition ses ressources est assez vieille. En effet, dès le début des années 1960, la notion de large infrastructure dont les ressources sont partagées dans le temps apparait [18]. Selon le document [71], l'appellation *cloud* date des années 90, en référence aux installations de télécommunication capables de répartir la charge. C'est avec l'arrivée des projets comme OpenNebula [69] ou OpenStack [70] que le Cloud Computing se démocratise. Depuis lors, de nombreuses définitions du Cloud ont été données, mais la littérature s'accorde désormais sur la définition proposée par le NIST (National Institute of Standards and Technology).

Le NIST propose dans [55] une définition communément acceptée par la communauté scientifique. Celle-ci définit le Cloud Computing comme « *un modèle permettant l'accès à un réseau de télécommunication, à la demande et en libre-service, à des ressources partagées et configurables* ». La définition du NIST propose également une description des services proposés par le Cloud, que nous détaillons dans la prochaine section.

Déclinaisons du Cloud Computing

Le Cloud Computing, selon la définition du NIST, est une architecture qui propose des ressources de manière configurable. Autrement dit, le service proposé s'accorde en fonction des besoins de l'utilisateur. Nous présentons dans la suite de cette partie les différentes déclinaisons de services identifiées dans [55].

Software as a Service (SaaS) Ce service propose à l'utilisateur de se reposer sur l'application déployée sur l'infrastructure. Ainsi, l'utilisateur ne doit pas configurer ou administrer les couches réseaux, le stockage, le système d'exploitation ou l'application.

Platform as a Service (PaaS) Ce service consiste à laisser l'utilisateur déployer sa propre application sur l'infrastructure. À l'instar du SaaS, l'utilisateur ne doit pas configurer ou administrer les couches réseaux, le stockage ou le système d'exploitation. Il doit cependant contrôler le déploiement, la configuration et le bon usage de son application.

Infrastructure as a Service (IaaS) Dans cette configuration de service, l'utilisateur a la liberté de déployer n'importe quelle application ou système d'exploitation sur l'infrastructure. Il n'a pas de contrôle sur les couches sous-jacentes de l'infrastructure, mais peut administrer le système d'exploitation, l'espace disque et la couche réseau (de manière modérée).

Modèles de déploiement

La définition du NIST [55] est complétée en détaillant les différents modèles de déploiement du Cloud Computing. Ces modèles désignent les différents publics visés pour une infrastructure de type Cloud Computing. Nous les présentons dans les paragraphes suivants.

Cloud privé L'infrastructure est réservée exclusivement aux personnes d'une entreprise ou organisation. L'administration, le contrôle et l'utilisation sont réservés aux membres de cette entreprise.

Cloud communautaire L'infrastructure appartient à une communauté d'organisations, qui partagent des objectifs (politique de sécurité, spécifications matérielles ou logicielles, etc.). L'administration, le contrôle et l'utilisation sont réservés aux membres de ces organisations.

Cloud public L'infrastructure appartient à une entité dans le but d'être utilisée par le grand public. Les objectifs de l'entité peuvent être académiques, pécuniers ou gouvernementaux. Dans le cas des fournisseurs de Cloud publics, les clients payent pour pouvoir accéder aux services proposés. L'administration, le contrôle et l'utilisation sont partagés entre le propriétaire de l'infrastructure et l'utilisateur.

Cloud hybride Un Cloud hybride est la composition de deux ou plus des types de Cloud précédemment introduits. Des interfaces standardisées et des technologies propriétaires existent pour faire coexister les différents types de Cloud.

Dans nos travaux, nous ne considérons que les Cloud dits « *publics* », qui sont plus sujets à des attaques, parce qu'il est nécessaire de contrôler à la fois les communications provenant de l'extérieur du réseau mais également à l'intérieur. Par ailleurs, la versatilité des clients est un verrou technologique. Nous détaillons ces verrous dans le Chapitre 3, qui présente la problématique de nos travaux.

2.2.2 La sécurité du Cloud

Le Cloud Computing est une infrastructure complexe, qui délivre des ressources selon les besoins de ses utilisateurs. Composée de machines virtuelles, de machines physiques, de composants réseaux variés, qu'en est-il de sa sécurité ? C'est à cette question que nous répondons dans cette partie. Tout d'abord, nous étudions les attaques réalisées sur le Cloud. Ensuite, nous présentons les différentes solutions utilisées dans le Cloud, puis nous nous intéressons aux faiblesses du Cloud inhérentes à sa structure.

Lorsque nous parlons de sécurité du Cloud dans ce mémoire, nous considérons uniquement les attaques dont le vecteur d'attaque est le réseau. Nous ne parlerons pas par conséquent d'attaques liées à la mémoire partagée, aux attaques par canal auxiliaire [4] ou encore de filoutage [5].

4. *Side-channel attack* en anglo-saxon
5. *phishing* en anglo-saxon

Les attaques réalisées sur le Cloud

Les attaques réalisées sur le Cloud sont nombreuses, nous les classons dans cette partie en fonction de la source et de la cible de l'attaque. Nous dénombrons trois types d'attaques possibles : les attaques provenant de l'extérieur ciblant des hôtes internes au réseau, les attaques provenant de l'intérieur ciblant des hôtes externes et pour finir les attaques internes au Cloud. Nous détaillons ces trois types d'attaques dans les paragraphes suivants tout en donnant des exemples d'attaques.

Bien que nous distinguons trois catégories d'attaques, il faut noter que ces attaques sont complémentaires. Par exemple, un attaquant débutera tout d'abord par tenter de s'infiltrer dans le Cloud. Pour cela, son attaque sera dirigée depuis l'extérieur vers l'intérieur du Cloud. Une fois qu'il a infecté un hôte, il essayera alors de s'attaquer à d'autres machines, internes ou non au Cloud Computing. La majorité des attaques d'ampleur est une combinaison de ces différents types d'attaques.

Le Cloud, cible d'attaques informatiques [59, 41] Le Cloud Computing, outre les services qu'il propose à ses utilisateurs, peut être considéré comme un centre de calcul. En effet, composé de machines physiques, virtuelles et de composants réseau, il s'agit d'un réseau de télécommunication usuel. Et tout comme les *data centers*, le Cloud est la cible d'attaques informatiques provenant de l'extérieur : propagation de vers, scan de ports, attaque de services réseaux, etc.

Réalisation d'attaques grâce au Cloud [58, 12] Nous avons présenté le Cloud dans la section précédente comme une infrastructure distribuant des ressources. Ces ressources considérables sont la proie des pirates, qui cherchent à les utiliser à leurs profits.

Attaques internes dans le Cloud [17, 5] Les motivations principales des pirates informatiques sont le profit financier, l'élimination de la concurrence ou encore la manipulation des tiers ou de leurs ressources. Une fois que ces individus sont infiltrés dans le Cloud (grâce à une attaque précédente ou en se faisant passer pour un utilisateur légitime), ils ont le contrôle de ressources puissantes, qu'ils peuvent utiliser pour atteindre leurs objectifs. Dans certains cas, la cible peut se situer directement sur le réseau du Cloud, nous parlerons alors d'attaques internes. Ces attaques proviennent des hôtes internes au réseau et ciblent des machines du Cloud. Nous verrons dans le Chapitre 3 que ces attaques internes sont particulièrement dangereuses.

Les solutions de sécurité utilisées dans le Cloud

Comme nous l'avons mentionné précédemment, le Cloud Computing est composé de machines physiques et de machines virtuelles. L'interconnexion entre ces machines est assuré par des composants réseau, tels que des routeurs ou des commutateurs. Dans cette section, nous discutons

des solutions de sécurité utilisées dans le Cloud, pour sécuriser l'ensemble de ces éléments.

Avant de présenter ces solutions, il est nécessaire de rappeler les objectifs de ces systèmes. Ils doivent assurer la « *sécurité* » du Cloud Computing, c'est-à-dire assurer un ensemble de propriétés [106]. Voici ces propriétés :

Disponibilité L'ensemble des services du Cloud Computing (les applications et son infrastructure) doit être accessible aux utilisateurs à tout moment.

Confidentialité Les données confidentielles [43] (informations relatives aux entreprises, coordonnées bancaires, brevets, etc.) des utilisateurs restent secrètes. Autrement dit, la politique de sécurité définit explicitement les usagers ayant la permission d'accéder à ces informations.

Intégrité Les données ne sont pas modifiées ou effacées par des utilisateurs non autorisés. Le système doit être capable de vérifier que les données n'ont pas été altérées durant une communication.

Contrôle L'action de réguler les ressources du Cloud Computing, dont les applications et l'infrastructure.

Authentification Lorsqu'un utilisateur agit sur les services mis en place, le système est capable de garantir que l'utilisateur est bien celui qu'il prétend être.

Non-répudiation Lorsqu'un utilisateur agit sur les services, il ne lui est pas possible de nier d'avoir réalisé cette action.

Puisque nous nous intéressons aux attaques dont le vecteur d'attaque est le réseau, les solutions que nous présenterons n'assureront pas toutes les propriétés ci-dessus. Les solutions utilisées dans le Cloud Computing sont les firewalls, les IDS et les IPS [60]. Par exemple, le fournisseur Amazon propose EC2 (Elastic Compute Cloud) [1], sécurisé par des firewalls. L'isolation des machines virtuelles est assurée par l'hyperviseur. Le client peut également configurer des solutions de sécurité déployées sur son instance.

La sécurité du Cloud, un frein à son adoption massive

Avec la popularité croissante du Cloud, sa sécurité est d'autant plus préoccupante et peut même retarder son adoption massive par certaines entreprises exigeantes. Dès 2010, une étude [89] rapporte que 62% des petites et moyennes entreprises ne désiraient pas passer au Cloud Computing pour des raisons de sécurité. Des chercheurs [3], des industriels [82] ou des organisations gouvernementales [28] partagent les mêmes inquiétudes.

L'étude du Ponemon Institute [43] corrobore ce sentiment d'insécurité dans le Cloud Computing. Leurs travaux se basent sur l'étude des services de plus de cents fournisseurs de Cloud Computing. Selon l'article, les fournisseurs de Cloud sont globalement assez peu confiants (à hauteur de 60%) quant au niveau de sécurité proposé, qu'il s'agisse des applications ou de l'infrastructure déployée. D'autres conclusions alarmantes sont tirées dans ce document. Notamment,

les fournisseurs accordent peu d'importance à la sécurité de leurs services et allouent peu de moyens pour la sécurité (humainement ou techniquement). Pour finir, ils estiment que la sécurité n'est pas la motivation principale des clients et que de ce fait, ce sont aux clients d'assurer la sécurité du service utilisé.

Les faiblesses inhérentes au modèle du Cloud

Dans cette section, nous discutons des faiblesses du Cloud, amenées avec son modèle de fonctionnenement. Nous considérons pour cette partie uniquement les clouds dits « *publics* », c'est-à-dire les infrastructures proposant au grand public de louer les ressources du Cloud.

Chen et al. [14] s'intéressent aux nouvelles problématiques en sécurité dans leur article *What's New About Cloud Computing Security ?*. Ils proposent une liste des problématiques, en les classant dans deux catégories distinctes. La première catégorie concerne les problématiques rencontrées dans le Cloud mais qui ont déjà été ciblées dans la littérature. Parmi ces problématiques existantes transposées au contexte du Cloud, on retrouve le filoutage, l'indisponibilité de services, la perte de données, la faiblesse des mots de passe, ou les hôtes infectés faisant partie de botnets. La deuxième catégorie correspond aux nouvelles problématiques soulevées par le Cloud. Parmi ces problématiques, la plupart existe déjà, mais leur importance est décuplée dans le cadre du Cloud. Nous nous intéressons à ces problématiques dans les paragraphes suivants.

Pour les pirates (également appelés *black hats* en anglo-saxon), le Cloud Computing est un modèle présentant de très bonnes propriétés : ressources considérables, à la demande et peu chère. C'est pourquoi ces pirates utilisent le Cloud, notamment pour casser des clés de chiffrement [58, 11] ou pour créer des botnets [40]. D'autre part, puisque le Cloud Computing se repose sur un ensemble de ressources partagées, des attaques peuvent avoir lieu sur les canaux de communication communs. Cette problématique n'est pas nouvelle, mais son impact est important au vue des objectifs de sécurité fixés. Notamment, parmi ces attaques, nous pouvons citer les fuites d'informations entre plusieurs machines virtuelles hébergées sur le même hôte physique [75] ou l'écoute passive du réseau pour en déduire des informations [86].

Une autre faiblesse référencée par Modi et al. [60] est l'attaque interne au réseau. Dans ce schéma d'attaque, l'utilisateur mal intentionné peut réaliser des attaques depuis le Cloud en ciblant des machines du même réseau. Ce type d'attaque a été démontré lors de la conférence DefCon 18 par Bryan and Anderson [12], durant laquelle ils ont réalisé une attaque DDoS depuis le Cloud EC2 d'Amazon, ciblant une autre de leur machine virtuelle, sur le même réseau. Cette démonstration, réalisée durant la conférence, montre les faiblesses de la structure du Cloud. Au moment de la conférence, les équipements de sécurité du réseau ne pouvaient pas détecter des attaques sur le même réseau, parce que le trafic réseau ne circulait pas au travers de ces équipements.

2.3 Les langages dédiés à la sécurité réseau

Dans la première section de cet état de l'art, nous avons étudié les solutions de sécurité réseau, leurs méthodes de détection et les solutions de sécurité collaboratives. Parmi ces méthodes de détection, nous avons présenté la détection par signatures, qui consiste à décrire une attaque dans un format particulier et de comparer les événements à ces signatures. Les solutions basées sur ce type de détection sont alors accompagnées d'un fichier qui contient l'ensemble des signatures. Ces signatures sont décrites grâce à un format spécifique, que l'on appelle également *langage dédié*. Les autres méthodes de détection utilisent également des fichiers de signatures, apparentés à ces fichiers de configuration. Nous présentons dans cette section ces langages dédiés et comment ils sont utilisés dans la littérature du domaine de la sécurité. Nous commencerons tout d'abord par une définition de ces langages, puis nous présentons les solutions existantes. Nous finissons enfin par une étude des méthodes d'évaluation des langages dédiés.

2.3.1 Définition

Mernik et al. [57] définissent les langages dédiés (*Domain Specific Language* ou DSL en anglais) comme des langages élaborés pour un domaine d'application spécifique. Salgueiro [80] complète cette définition en ajoutant qu'il s'agit de petits langages, très expressif et ciblant un domaine d'application particulier, contrairement aux langages dit « *généraux* », qui sont conçus pour être le plus permissif possible.

L'élaboration de tels langages est motivée par le manque de langage capable de décrire des concepts très spécifiques de manière simple. Notamment, un DSL peut permettre d'exprimer des notions qui ne sont pas exprimables dans les langages généraux. De plus, les DSL peuvent permettre de faciliter l'analyse, l'optimisation ou la parallélisation du problème exprimé, grâce à des notions très spécifiques au domaine ciblé. Cela permet à des utilisateurs qui ne sont pas experts du domaine d'écrire des programmes valides de manière efficace.

Le développement de ces DSL est jugé complexe, car il est nécessaire à la fois d'élaborer un langage mais aussi d'étudier un domaine très précisément afin d'en extraire des notions spécifiques ainsi que leur sémantique. À l'instar des langages généraux, il existe des patrons de conception pour l'élaboration de ces langages, tels que l'automatisation de tâche ou la représentation de structure de données.

Les langages dédiés interviennent dans de nombreux domaines, généralement conçus pour faciliter le travail de développement. Il y a un certain nombre d'outils que l'on utilise tous les jours qui sont constitués de langages dédiés. Voici quelques exemples :

— LaTeX, un langage spécialisé pour la rédaction de documents scientifiques ;

— Make, un outil permettant de construire automatiquement des fichiers, en utilisant une notion de dépendances entre les fichiers ;

— SQL, un langage qui permet de décrire des requêtes dans une base de données ;

— VHDL, un langage permettant de décrire le schéma électronique d'un composant FPGA de manière fonctionnelle.

Salgueiro [80] distingue deux types de DSL : les DSL *internes* et les DSL *externes*. Les langages internes sont conçus et intégrés dans un langage général, sous forme de librairie de manière à étendre le langage général et ajouter des notions du domaine ciblé. Quant aux langages externes, ils sont réalisés depuis zéro et fonctionnent indépendamment d'un quelconque langage général.

2.3.2 Langages dédiés à la détection d'intrusions

Dans cette section, nous nous intéressons aux langages consacrés à la détection d'intrusions. Ces langages sont proposés dans le but de décrire les moyens de détecter les attaques en cours sur un réseau donné. Ce type de langage permet de configurer les solutions de sécurité et de faciliter leur mise à jour. La majorité de ces langages sont des langages externes, mais des langages internes existent. Nous étudions dans les prochains paragraphes les solutions existantes de la littérature.

Snort

Snort [7, 76], reconnu comme une des solutions de référence dans la littérature, est un projet open-source qui propose un IDS multi-plateforme. Basé sur `libpcap` [46], Snort est capable d'analyser en temps réel le trafic réseau et alerte les administrateurs en cas d'intrusion.

Snort est basé sur un ensemble de modules, qui permettent de réaliser des opérations sur les paquets à inspecter, dont par exemple le module *Stream* qui est responsable de la recomposition des paquets fragmentés. Snort se repose également sur un moteur de filtrage par motif[6] qui recherche des motifs connus dans les paquets réseau. Ces motifs sont décrits au travers des signatures. Pour décrire les signatures des intrusions à détecter, Snort fournit un langage dédié dans lequel une signature correspond à un ensemble de conditions à tester et à l'action à réaliser si les conditions sont réunies. Grâce à son importante communauté de contributeurs, Snort a un ensemble de règles important et exhaustif sur la plupart des attaques et intrusions connues.

Le langage de Snort est déclaratif : une règle correspond aux tests à réaliser sur chaque paquet et à l'action associée en cas d'intrusion détectée. Le Listing 2.1 propose un exemple de signature Snort, qui génère une alerte pour chaque paquet réseau provenant du réseau externe vers le réseau interne, dirigé vers le port 80, pour toute connexion TCP ouverte.

6. *Pattern-matching* en anglo-saxon

Listing 2.1– Exemple de règle Snort

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 80 \
    (msg:"HTTP packet"; flow:to_server,established;)
```

Le langage proposé par Snort permet de décrire un grand nombre d'attaques, mais a aussi des désavantages. Notamment il est très difficile d'écrire des règles qui analysent plusieurs paquets distincts et qui se produisent à deux instants différents. En effet, une règle Snort décrit l'analyse d'un paquet réseau et il est assez laborieux de s'abstraire de cette approche [7]. D'autre part, les analyses statistiques restent limitées et il est impossible d'exprimer dans le langage un moyen de conserver des données sur l'état courant du système. Pour finir, de manière simplifiée, ce langage permet de décrire les attaques réseau comme une séquence d'octets, qui est recherchée dans le contenu des paquets. La description des attaques se révèle assez facile, mais reste très spécifique. Le risque est alors de vouloir généraliser une signature et ainsi de provoquer un plus grand nombre de faux-positifs.

Bro

Bro est également une solution populaire dans la littérature. Proposé par Vern Paxson dans [64], Bro est un IDS open-source qui se repose à la fois sur la détection d'anomalies et la détection par signatures. La politique de sécurité est écrite dans un fichier de configuration, grâce au langage proposé par Bro.

Contrairement aux signatures de Snort qui sont décrites de manière déclarative, les signatures de Bro sont décrites de manière impérative, tel un script. Les concepts de la programmation impérative sont utilisées dans ce langage, à savoir : fonctions, structures de données, variables typées ou constantes. Le Listing 2.2 propose un exemple d'utilisation du langage de Bro. Dans cet exemple, l'événement `arp_reply` met à jour l'association entre adresse MAC et adresse IP.

Listing 2.2– Exemple de script Bro

```
event arp_reply(mac_src: string, mac_dst: string, SPA: addr, SHA: string, TPA: addr, THA: string)
{
    local ip = SPA;
    local mac = mac_src;

    if (ip !in ip_to_mac)
        ip_to_mac[ip] = mac;
}
```

7. l'option `flowbits` permettrait d'écrire de telles règles, mais il est reconnu par la communauté et la littérature que cette méthode d'écriture est contre-intuitive et susceptible de causer des erreurs [80]

Lambda

Proposé par Cuppens and Ortalo [23], Lambda est un langage déclaratif de description logique d'attaques. Une description d'attaque est décomposée en quatre parties :

- la **pré-condition** définit l'état du système requis pour que l'attaque soit réalisable ;
- la **post-condition** définit l'état du système après que l'attaque ait eu lieu ;
- la **détection** décrit les différentes alertes attendues lors de la phase de détection ;
- la **vérification** correspond à une suite de conditions qui permet d'attester que l'attaque a bien eu lieu.

Le Listing 2.3 propose un exemple d'utilisation de Lambda, dans lequel on cherche à détecter un scan de port. L'écriture de ces règles permet d'établir des scénarios d'attaques, qui sont liés entre eux grâce aux pré et post-conditions. Notamment, il est tout à fait imaginable qu'une autre règle de détection nécessite comme pré-condition qu'une attaque sache qu'un port TCP est ouvert, qui se trouve être la post-condition de la règle définie dans le Listing 2.3.

Listing 2.3– Détection d'un scan de port en Lambda

```
1  attack scan−port(Agent,Host,Port)
2  pre: open(Host,Port)
3               – Port is open on Host
4  detection: classification(Alert,'TCP−Scan')
5               – the alert classification is 'TCP−Scan'
6               ^ source(Alert,Agent)
7               – the source in alert is Agent
8               ^ target(Alert,Host)
9               – the target in alert is Host
10              ^ target service port(Alert,Port)
11              – the scanned port is Port
12 post: knows(Agent,open(Host,Port))
13              – Agent knows that Port is open
14 verification: true
15              – always true
```

Lambda a été ensuite complété, au travers de DIAMS [21], pour pouvoir également contenir dans la description un moyen de définir les réactions possibles à la suite de la détection d'une attaque. Pour cela, la partie supplémentaire **counter-measure** permet de définir la réaction à adopter.

NeMode

Proposé au travers de plusieurs articles [81, 79, 78], NeMode est un langage dédié permettant de décrire des attaques réseau de manière déclarative. Les règles décrites dans ce langage se

basent sur la programmation par contrainte, un paradigme de programmation où l'utilisateur décrit de manière logique les étapes d'une attaque réseau.

Le Listing 2.4 propose un exemple d'utilisation de NeMode, au travers duquel le système cherche à détecter l'attaque SYN Flood. Cette attaque consiste à ouvrir de nombreuses connexions TCP. Une méthode naïve pour la détecter consiste à compter le nombre d'ouvertures de connexion. Si ce nombre dépasse un certain seuil pour une période donnée, le système en déduit qu'il y a une attaque en cours. Dans cet exemple, nous considérons qu'une attaque est en cours s'il y a plus de 30 ouvertures de connexion sur une période de 500 microsecondes.

Listing 2.4– Détection d'une attaque SYN Flood

```
1 syn_flood {
2     C = { tcp_packet(A),
3           syn(A), nak(A) },
4     R := repeat(30,C),
5     max_interval(R) < usecs(500)
6 } => {
7     alert('SYN flood attack attempt')
8 };
```

STATL

Se basant sur la notion d'automate à états, STATL [27] est un langage dédié permettant de décrire une attaque comme une séquence d'actions. Ce scénario peut alors être utilisé pour détecter les intrusions. La séquence d'actions, qui décrit alors un automate à états, peut exprimer des schémas d'attaques réseau ou système. Le langage propose des mécanismes semblables à un langage impératif, ainsi que des concepts spécifiques à la méthode de détection adoptée par STATL : scénario, état ou encore transition.

PLAN-P et binpac

Bien que *binpac* ne soit pas un langage dédié à la sécurité réseau, les travaux de Pang et al. [63] proposent un langage de description des protocoles réseau afin de générer des analyseurs de trafic réseau. Puisque le développement d'analyseur de protocole réseau était long, fastidieux et susceptible d'introduire des erreurs, ils proposent un langage abstrait de représentation des données. Grâce à des mécanismes de structure de données ou d'expression régulière, binpac est capable de décrire des protocoles réseau binaires ou textes, tels que HTTP, FTP ou DNS.

Proposé par Muller et al. [62], PLAN-P est une solution qui a les mêmes ambitions. Elle souhaite faciliter le développement de drivers réseau, dans l'optique d'apporter des propriétés de fiabilité, de sécurité, de portabilité et de performance. Dans sa syntaxe, le langage se rapproche d'un langage impératif : les protocoles sont décrits grâce à des structures de données, des

fonctions, des variables, etc.

Appliquées à la sécurité réseau, ces deux solutions auraient le principal désavantage d'appliquer uniquement de la détection d'anomalies. En effet, la description de protocole permet de déterminer si un paquet réseau est correct vis à vis de la norme. Autrement dit, ces solutions pourraient conclure qu'un paquet est malveillant car il ne s'applique pas au modèle défini. Toutefois, elles seraient incapables de déterminer qu'un paquet réseau serait malveillant si celui-ci utilisait des failles du protocole : le paquet serait considéré légitime par ce type de solution puisqu'il serait décrit dans le modèle.

Zebra

Zebra est une solution qui s'intéresse aux mêmes problématiques que PLAN-P ou binpac : la construction de programmes qui décomposent et analysent le contenu du trafic réseau. Extension du projet Zebu [13], Zebra est proposé par Mercadal et al. [56]. Il s'intéresse à la décomposition de protocoles réseau texte, tels que HTTP, SMTP ou encore SIP. Les auteurs constatent que cette tâche est désormais déléguée à des composants hardware, tels que des ASIC ou des FPGA. Cela complique le développement de ces drivers, puisqu'il est nécessaire d'être expert à la fois en télécommunication, en système et en développement hardware. Pour parer à cette difficulté, Zebra propose un langage dédié permettant de décrire les protocoles réseau ciblés. Grâce à cette description, le compilateur génère le code associé, spécialisé pour le matériel visé.

Appliqué au domaine de la sécurité réseau, Zebra présente les mêmes désavantages que PLAN-P ou binpac. De plus, la reconfiguration de ces matériels ne serait pas envisageable étant donné le temps de compilation et de déploiement, bien qu'il existe des mécanismes de reconfiguration partielle.

2.3.3 Méthodes d'évaluation des langages dédiés

Dans nos travaux, nous proposons un langage dédié, que nous cherchons à positionner par rapport aux solutions existantes. C'est pourquoi nous étudions dans cette section les méthodes d'évaluation utilisées dans la littérature [72, 80, 21]. Les prochains paragraphes présentent ces méthodes.

Expressivité La première méthode d'évaluation est la capacité du langage à exprimer les concepts du domaine étudié. Puisque l'objectif principal d'un langage dédié est de « *capturer l'essence* » d'un domaine, on peut attendre de ce langage qu'il soit capable d'exprimer l'ensemble des notions du domaine. Ce type d'évaluation consiste donc à établir une liste des notions que le langage est capable ou non d'exprimer et de le comparer à l'existant.

Robustesse Une autre méthode d'évaluation est d'établir le niveau de robustesse de l'application. L'objectif d'un langage dédié est d'être spécifique à un domaine, par conséquent d'être capable de détecter les situations posant problème ou améliorables. C'est pourquoi une méthode d'évaluation consiste à confronter le langage face à des situations pour lesquelles il devrait réagir et quantifier le nombre de réactions positives.

Performance La dernière métrique que l'on étudiera dans nos travaux est la performance. Dans notre cas, il s'agit d'évaluer si l'interprétation d'une politique de sécurité réalisée par le langage est performante. En ce qui concerne les métriques de performances, nous utiliserons les métriques proposées dans la Section 2.1.5 qui se concentrent sur l'évaluation des performances des IDS.

3

Problématique

Dans ce chapitre, nous établissons la problématique étudiée dans ces travaux. Dans un premier temps, nous présentons l'architecture que nous ciblons dans ce mémoire, à savoir le Cloud Computing. Dans cette partie, nous distinguons les verrous scientifiques et technologiques que nous souhaitons lever. Ensuite, nous mettons en avant que cette infrastructure présente des faiblesses, au travers d'expérimentations qui se concentrent sur des attaques distribuées. Pour finir, nous concluons ce chapitre en présentant Discus, la solution que nous proposons pour répondre à cette problématique.

3.1 Sécurité du Cloud

Dans cette section, nous discutons de la sécurité du Cloud Computing et des verrous scientifiques et technologiques existants. Comme nous le mentionnons dans le précédent chapitre, le Cloud Computing est une composition de nombreux équipements : équipements de sécurité, équipements réseau et de nombreuses machines physiques. Ces dernières hébergent des machines virtuelles volatiles, qui peuvent être migrées ou interrompues à tout moment.

Cette structuration est un héritage des centres de données, la majorité des problématiques rencontrées dans le Cloud a donc déjà été abordée dans la littérature. Toutefois, de nouveaux enjeux sont apparus avec le Cloud, dû au fait que les ressources sont partagées entre de nombreux utilisateurs, dont certains peuvent être malicieux. C'est pourquoi de nouveaux moyens sont nécessaires et doivent proposer des propriétés de sécurité sur cette plateforme. Les propriétés ciblées sont : la disponibilité des services, la confidentialité, l'intégrité, le contrôle, l'authentification et la non-répudiation [1].

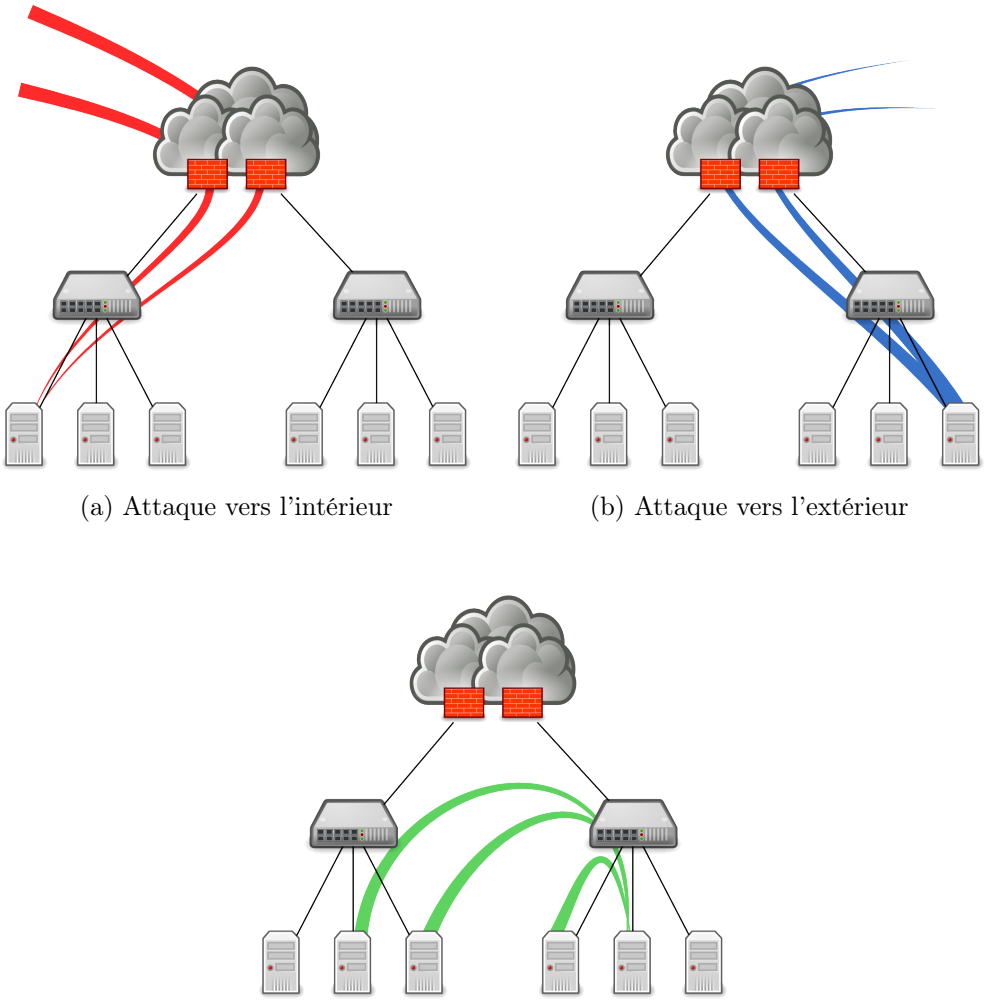
Nous listons dans les paragraphes suivants les problématiques et verrous que nous ciblons dans nos travaux. Nous rappelons au lecteur que nos travaux s'intéressent à des structures publiques, où un utilisateur *lambda* peut louer des services sur une période donnée. D'autre part, notre objectif est de pouvoir détecter les attaques dont le vecteur de communication est le réseau, c'est pourquoi nous ne traiterons pas dans cette section des attaques logicielles ou physiques.

3.1.1 Détection d'attaques dirigées

Les solutions de sécurité usuelles considèrent à tort que deux réseaux existent : le réseau à protéger et le reste du monde. Cette hypothèse, généralement adoptée par les équipements placés en bordure comme les firewalls, n'est plus correcte avec le Cloud. En effet, cela implique que le système ait confiance en le réseau interne et qu'aucun de ses utilisateurs ne soit malicieux. Nous avons vu au travers de plusieurs articles [58, 11, 40] que cette hypothèse n'est pas valide dans le contexte du Cloud, puisque l'utilisateur malicieux peut être présent sur le réseau interne, soit en s'infiltrant dans le réseau en question, ou tout simplement en louant les services du Cloud de manière légitime.

1. Ces propriétés ont été définies dans la Section 2.2.2

Une attaque est dirigée depuis le poste attaquant vers le poste attaqué. La localisation de ces postes sur les différents réseaux est importante en ce qui concerne la détection des attaques. En effet, si nous nous basons sur l'hypothèse initiale que le poste malicieux est situé sur le réseau externe, une attaque dirigée depuis le Cloud ne serait pas détectée. Nous distinguons trois types de directions, illustrées dans la Figure 3.1.



(a) Attaque vers l'intérieur (b) Attaque vers l'extérieur

(c) Attaque interne

FIGURE 3.1 – Différentes directions des attaques

Dans la Figure 3.1, nous avons représenté les différentes directions d'attaques que nous distinguons dans nos travaux. La Figure 3.1a illustre une attaque traditionnelle, dirigée depuis l'extérieur du réseau vers l'intérieur du réseau. Les solutions de sécurité basiques se concentrent sur la détection de ce type d'attaque. Le second type d'attaque, illustré dans la Figure 3.1b, représente une attaque provenant de l'intérieur du réseau vers l'extérieur du réseau. Si l'on considère que les utilisateurs internes de notre réseau ne sont pas malicieux, il nous est alors impossible de détecter ce type d'attaque. Pour finir, la Figure 3.1 représente les attaques à l'intérieur même

d'un réseau. Ce type d'attaque est particulièrement efficace, car les flux de données ne traversent pas les solutions de sécurité. D'autre part, ces attaques internes peuvent se produire au sein même d'une grappe de machines, voire au sein d'une même machine physique, entre deux machines virtuelles. Il faut également noter que toutes ces attaques peuvent se produire de manière distribuée, c'est-à-dire qu'une attaque peut être dirigée vers plusieurs cibles et/ou provenir de plusieurs attaquants.

Dans le contexte du Cloud, il est nécessaire de proposer un moyen de détecter tous ces types d'attaques. Le verrou ici est de choisir la bonne granularité de placement des solutions de sécurité. Les attaques les plus difficiles à détecter sont les attaques internes, surtout quand elles se produisent de manière rapprochée. Idéalement, il serait donc souhaitable d'avoir au moins une solution de sécurité sur chaque connexion du réseau. Cela ne semble pas envisageable, pour des raisons de coût et de maintenance. En effet, il serait très coûteux de déployer des firewalls ou des IDS sur tous les liens réseau. Qui plus est, la configuration de ces équipements ne serait pas une tâche facile, puisque la structure du Cloud change en permanence. Il faut donc établir un compromis en ce qui concerne la présence de ces solutions sur le réseau. D'autre part, pour surveiller l'ensemble des postes, il sera nécessaire d'utiliser différentes solutions de sécurité. Nous traitons de la problématique de l'hétérogénéité des solutions de sécurité dans le prochain paragraphe.

3.1.2 Configuration de solutions de sécurité hétérogènes

Pour surveiller un réseau de la taille du Cloud, il sera nécessaire de conjuguer différentes solutions de sécurité (firewalls, IDS ou IPS) qui peuvent être implantées physiquement sur le réseau, ou incorporées logiciellement sur les équipements à surveiller.

L'hétérogénéité de ces équipements est une source de problèmes. Tout d'abord, il est nécessaire de les implanter sur le réseau à protéger. Cette série de manipulations peut se révéler complexe sur un réseau de type Cloud, puisque la topologie physique est souvent inconnue. Autrement dit, le propriétaire d'un Cloud n'est généralement pas capable de donner une description de la structure de son équipement. D'autre part, il est nécessaire de configurer l'ensemble de ces solutions. La configuration de l'ensemble du parc de solutions de sécurité est laborieuse, longue et susceptible de provoquer des erreurs. En effet, puisqu'il est nécessaire de combiner différents types d'équipements de sécurité, il faut configurer chacun de ces équipements avec ses propres outils, qui peuvent utiliser des paradigmes et des moteurs de détections différents. C'est effectivement ce que nous avons mis en lumière dans l'état de l'art, lorsque nous nous sommes intéressés aux langages dédiés ou aux méthodes de détection.

En plus de se reposer sur un grand nombre de composants de la sécurité, l'état de l'art nous indique qu'il est possible de rendre ces composants coopératifs. Cette approche permet de faire en sorte qu'un équipement de sécurité puisse collaborer avec un autre équipement, de manière à détecter plus efficacement et rapidement les attaques. Puisque les solutions de sécurité sont hétérogènes, il faut trouver un moyen pour les faire collaborer. Nous aborderons

cette problématique dans la section 3.1.5.

Pour finir, un certain nombre d'enjeux est soulevé par une architecture distribuée et hétérogènes, tels que :

— la capacité à configurer et à mettre à jour à la volée les solutions sans pour autant couper les services déployés ;
— la capacité à détecter qu'une solution est en panne et être capable de basculer les services de cette solution vers une autre qui est opérationnelle ;
— la capacité à détecter qu'une solution a été compromise et cherche délibérément à altérer le comportement nominal de la solution en sa globalité.

3.1.3 Volatilité des services déployés

Le Cloud est une structure qui est modifiée en permanence, que ce soit à l'initiative des utilisateurs ou du Cloud Computing. En effet, les machines virtuelles allouées aux utilisateurs démarrent, s'arrêtent et s'interrompent à la demande de l'utilisateur. Il en est de même pour les machines virtuelles qui peuvent être migrées « *à chaud* », c'est-à-dire durant l'exécution d'un service.

La configuration des solutions de sécurité n'est donc pas chose aisée dans un contexte constamment changeant. En cas de modification de la structure du Cloud (démarrage, interruption, arrêt ou migration d'une VM), la politique de sécurité doit être modifiée en conséquence, de manière transparente pour l'utilisateur. Nos travaux ne s'intéressent malheureusement pas à cette problématique. Des travaux en cours [36], portant sur la reconfiguration à chaud de la politique de sécurité, pourraient être une excellente piste pour compléter notre approche.

3.1.4 Implication des différents acteurs dans l'écriture de la politique de sécurité

Nous avons défini la politique de sécurité comme étant une suite de règles qui statue ce qui est permis ou non par le système. Cette politique est généralement écrite par un nombre restreint de personnes dans le cadre de centre de données, ce qui n'est pas le cas pour le Cloud. En effet, de nombreux acteurs sont impliqués dans l'établissement de cette politique ; nous les détaillons dans les prochains paragraphes.

Tout d'abord, de manière assez classique, les **administrateurs système, réseau et sécurité** interviennent pour établir une infrastructure qui propose les services attendus. Communs aux centres de données, la tâche de ces acteurs est généralement très spécifique : certains administrateurs seront spécialisés dans l'aspect sécurité alors que d'autres se concentreront sur l'aspect logiciel et déploiement. Le rôle de l'administrateur de sécurité est de définir un ensemble de règles de sécurité spécifique à l'infrastructure.

Également, d'autres acteurs étrangers à la structure à surveiller interviennent dans l'écriture de cette politique de sécurité. Le **développeur de solutions de sécurité** conçoit un service (logiciel ou matériel) qui doit être capable de s'adapter à un grand nombre d'infrastructures différentes. Il peut intervenir dans l'écriture de la politique en proposant des règles générales de sécurité, s'appliquant communément aux systèmes. L'**expert en sécurité** peut enrichir une base de données de règles de sécurité. Parfois bénévole, à l'instar de la communauté Snort, ces experts proposent des règles généralistes.

L'**utilisateur** peut également interagir dans la configuration de ces équipements de deux manières différentes. Premièrement, la sécurité peut être vue comme un service payant. La sécurité est alors modulable suivant la somme que l'utilisateur est prêt à dépenser. La configuration des solutions est alors différentes pour chacun des utilisateurs. Deuxièmement, il est aussi envisageable que l'utilisateur intervienne de manière active dans la configuration de la surveillance de ses services. Qu'il s'agisse de l'installation de solutions de sécurité supplémentaires ou de la rédaction de règles de sécurité propres à son service, l'utilisateur peut compléter la sécurité offerte par le fournisseur.

Ce grand nombre d'acteurs est une source de problèmes, car il est nécessaire que chacune de ces personnes puisse s'exprimer et être comprise par l'ensemble des acteurs. C'est pourquoi il est nécessaire de proposer une base commune, sur laquelle chacun des participants pourra échafauder ses propres règles de sécurité.

3.1.5 Distribution efficace de la sécurité

Historiquement, les premières solutions de sécurité agissaient en un point du réseau, de manière individuelle, en analysant les événements locaux. Nous avons présenté dans l'état de l'art que la tendance actuelle est de distribuer ces solutions de sécurité, afin de s'adapter aux nouvelles architectures. La distribution permet de mener des analyses plus précises puisque les données traitées proviennent de l'ensemble du réseau surveillé. Également, cette distribution permet d'avoir une architecture plus robuste en cas de panne ou de dysfonctionnement d'un nœud du réseau. Toutefois, la distribution est également la source de questionnements dont nous discuterons dans les prochains paragraphes.

Tout d'abord, la distribution et la collaboration des solutions de sécurité implique une forme de communication entre ces éléments. Il est nécessaire d'identifier la nature et le mode de propagation de ces messages mais également de conserver des messages concis et peu réguliers, pour éviter d'avoir un impact important sur le trafic nominal de la structure surveillée. De plus, il est important d'établir la structuration de ces éléments et de spécifier les capacités de chacun. Pour finir, la sélection de la méthode de détection est également peu aisée car elle doit s'adapter à une structure qui est modifiée en permanence, elle doit offrir la capacité de reconfiguration rapide et être consciente de la structure physique et logique.

3.1.6 Synthèse

Dans cette section, nous avons mis en avant les différentes problématiques liées au Cloud Computing dans le contexte d'attaques transitant par le réseau. La majorité de ces problématiques a déjà été abordée dans la littérature, notamment dans le domaine des centres de données. La grandeur du réseau ciblé et son aspect volatil sontla source de problématiques que nous étudions dans ces travaux. Concernant la sécurité d'une telle infrastructure, une approche communément admise dans la littérature est de distribuer les solutions de sécurité, afin d'analyser finement les données et ainsi être capable de détecter les attaques efficacement et rapidement.

Dans ce cadre, deux problématiques majeures se dégagent. La première est la gestion de l'ensemble des composants de la structure de sécurité déployée. Il est nécessaire d'adapter la politique de sécurité en fonction du composant sécurité en question, des machines à surveiller, des clients et des services souscrits. Également, l'aspect volatile du Cloud contraint que la solution puisse rapidement se reconfigurer pour offrir le niveau de sécurité attendu. La deuxième problématique se concentre plutôt sur la méthode de détection à utiliser dans un tel cadre. Pour pouvoir détecter des attaques à large échelle, telles que les attaques distribuées, il est nécessaire de corréler des informations provenant de plusieurs endroits du réseau. Cette collaboration soulève des questions quant à la nature des échanges, leur source et leur forme.

Nos travaux se concentrent principalement sur la première problématique, à savoir la gestion du parc hétérogène de solutions de sécurité. Nous proposons néanmoins des débuts de pistes en ce qui concerne la collaboration de ces solutions. Nous vous présentons dans la Section 3.3 notre solution, Discus, qui se propose de répondre à ces verrous scientifiques. Avant cela, le lecteur trouvera dans la Section 3.2 les expérimentations préalables menées pour établir la dangerosité des attaques distribuées.

3.2 Attaques distribuées : des chiffres alarmants

Dans cette section, nous étudions l'impact des attaques distribuées sur des réseaux et leur efficacité par rapport à des attaques traditionnelles. Tout d'abord, nous rappelons au lecteur la nature des attaques distribuées et les motivations qui nous ont poussées à conduire des expérimentations, découpées en deux étapes. Dans un premier temps, nous quantifions l'impact des attaques distribuées classiques, qui traversent un seul point de passage qui analyse le trafic réseau. Ensuite, nous analysons le même type d'attaques sur une structure qui se rapproche des infrastructures réelles, avec plusieurs points d'entrée d'analyse.

3.2.1 Motivations

Les attaques distribuées consistent à morceler une attaque, de manière à réaliser l'attaque depuis plusieurs hôtes et/ou en ciblant plusieurs postes. Les objectifs de ces attaques sont mul-

tiples. Principalement, il s'agit d'un moyen simple d'éviter de se faire détecter par les solutions de sécurité. En effet, découper une attaque en « petits morceaux »permet d'attirer moins l'attention sur l'objectif réel de l'attaque. Cela permet également d'être moins facilement identifiable lorsque l'attaque provient de plusieurs machines attaquantes. L'utilisation d'attaques distribuées est courante dans les *botnets*, constitués de nombreuses machines *zombies* qui sont généralement des machines infectées sur Internet.

Ces attaques sont très souvent abordées dans la littérature, notamment au travers des vers informatique, des botnets ou des attaques de déni de services. Néanmoins, à notre connaissance, il n'existe pas d'étude qui se concentre sur la quantification de l'impact de ces attaques. C'est pourquoi nous avons mener une série d'expérimentations préalables, qui répondent à la question suivante : à quel point une attaque distribuée est plus efficace qu'une attaque traditionnelle ?

Pour répondre à cette question, nous avons réalisé deux types d'expériences, qui se concentrent chacune sur un aspect différent. La première série d'expériences s'intéressent à l'efficacité des attaques distribuées dans une structure réseau simpliste, où un seul nœud intermédiaire est responsable de l'analyse du trafic réseau. La deuxième série d'expériences se base sur une structure plus complexe, où différents nœuds intermédiaires existent entre les attaquants et les cibles de l'attaque. Ces expérimentations nous permettent alors d'évaluer l'efficacité des attaques distribuées. Nous parlons dans les études à suivre d'efficacité du point de vue de l'attaquant. Autrement dit, une attaque est plus efficace qu'une autre si elle se fait détecter moins souvent et moins rapidement.

3.2.2 Cas d'étude : le scan de ports

Après avoir étudié les attaques distribuées existantes, nous avons choisi de baser nos expérimentations sur le scan de ports distribué. Nous présentons dans cette section le scan de ports et les manières de distribuer une telle attaque.

Définition du scan de ports

Le balayage de ports, plus communément appelé scan de ports dans la littérature, est une attaque dont l'objectif est de déterminer l'état des services réseau d'une machine [48]. Il s'agit d'une séquence de paquets réseau, initiée par l'attaquant, qui permet de statuer si un port réseau est ouvert ou non. Également, cela permet parfois de déterminer des informations du système : système d'exploitation installé, version des applications, etc. Cette phase de reconnaissance est généralement suivie d'une phase d'attaque, durant laquelle l'individu malicieux se base sur les points d'entrée identifiés lors du scan de ports.

Le scan de ports peut être réalisé de plusieurs manières, en manipulant les différents champs des paquets TCP ou UDP. On peut distinguer deux types de scans de ports. Le premier, que l'on appellera scan de ports légitime, repose sur des échanges de paquets qui sont légitimes vis à vis de

la spécification du protocole. Par exemple, le scan de ports appelé « *Connect Scanning* » consiste à ouvrir une connexion TCP, ce qui est légitime. Le second type de scans de ports, dit illégitime, consiste à envoyer des paquets qui ne sont pas renseignés ou conformes selon la spécification. Par exemple, la technique « *XMAS scanning* » consiste à envoyer un paquet TCP qui contient un grand nombre de drapeaux levés (à la manière d'un arbre de Noël, d'où le nom de la technique), ce qui est anormal selon le protocole TCP. Ce type d'attaques peut être concluant lorsque la technique utilisée se repose sur une vulnérabilité ou que l'équipement réseau ne détecte pas ces cas anormaux. Nous discutons des méthodes de détection du scan de ports dans la prochaine section.

Détection de scan de ports

Dans le cadre de nos expérimentations, nous cherchons à quantifier l'efficacité des attaques distribuées. Nous avons indiqué qu'une attaque est efficace si elle parvient à ne pas se faire détecter ou retarder au maximum sa détection. Nous expliquons dans les prochains paragraphes les méthodes de détections usuelles pour l'attaque de scan de ports. Notre objet d'étude étant l'attaque distribuée en général, les explications de ces méthodes seront brèves mais concises, de manière à pouvoir interpréter les prochains résultats.

Pour les techniques de scan de ports légitimes, il reste assez difficile de détecter les attaques. En effet, la solution de sécurité doit déterminer si une séquence de paquets, légitime du point de vue du protocole, est malicieuse. Il est d'autant plus difficile de distinguer les scans de ports légitimes (recherche d'imprimante, réseau pair à pair, etc) des scans de ports malicieux. Pour détecter ce type de scans de ports, la première méthode consiste à dénombrer les événements et de conclure qu'une attaque est en cours une fois un certain seuil dépassé, pour une période de temps fixée. D'autres méthodes se basent sur des modèles mathématiques pour détecter les attaques. Notamment, la technique « *Threshold Random Walk* » [48] se base sur des statistiques pour évaluer si une attaque est en cours.

La détection de scans de ports illégitimes, c'est-à-dire qui ne sont pas conformes ou non renseignés dans la spécification, est généralement plus simple. Effectivement, il est possible de décrire ces paquets (selon les options ou drapeaux levés) au travers de signatures de sécurité. La solution de sécurité devra donc comparer les paquets courants à ces signatures et conclure qu'il y a une attaque lorsque la signature est détectée.

Distribution du scan de ports

Au travers de nos expérimentations, nous cherchons à distribuer des attaques réseaux, c'est-à-dire morceler l'attaque de manière à ne pas se faire détecter. Dans le cadre de nos mesures, la distribution aura lieu tant au niveau des attaquants que des cibles. En effet, nous avons utilisé de nombreux attaquants qui visent une ou plusieurs cibles. Nous présentons les méthodes utilisées pour distribuer l'attaque. Il faut noter que nos expérimentations se sont satisfaites de méthodes

de distribution simples et que des méthodes plus efficaces et sophistiquées existent. Toutefois, de très bons résultats ont été obtenus grâce à ces méthodes simplistes.

La distribution **naïve**, proposée par Kang et al. [49], consiste à utiliser un ensemble de nœuds attaquants, les uns après les autres. Pour cela, le premier nœud réalise l'attaque, une fois détecté par la solution de sécurité, le prochain nœud disponible continue l'attaque. Maitrisant l'architecture d'expérimentation, il nous est possible de savoir quand un nœud est détecté [2]. Nous pouvons espérer de cette méthode qu'elle soit linéaire : si un nœud peut scanner x ports sans être détecté, deux nœuds peuvent scanner $2x$ ports.

La distribution **parallèle** découpe l'attaque et distribue équitablement la charge de travail entre les nœuds. L'ensemble des nœuds attaquants démarre en même temps l'attaque et envoie par la suite les résultats au coordinateur. L'aspect parallèle peut générer des latences au niveau de l'analyse des paquets et donc créer un délai pour la détection de l'attaque.

La distribution **séquentielle** consiste à utiliser tour à tour chacun des nœuds attaquants, de manière à espacer deux utilisations du même nœud attaquant. Cette technique permet d'éviter la détection par seuil, où la solution de sécurité dénombre les événements suspects pour une période de temps fixée.

3.2.3 Comportement des solutions de sécurité dans un contexte distribué

Dans cette première étude [73], nous nous sommes intéressés à l'impact des attaques distribuées en répondant à la question suivante : « *À quel point une attaque distribuée est plus efficace qu'une attaque traditionnelle ?* ». Notre objectif était d'évaluer l'efficacité d'une attaque distribuée et de la comparer à une attaque traditionnelle. Nous parlons d'attaque traditionnelle lorsqu'un seul attaquant est utilisé. Dans les prochains paragraphes, nous considérons qu'une attaque est plus efficace qu'une autre si elle a été détectée moins rapidement, voire pas du tout.

Nous illustrons dans la Figure 3.2 la structure utilisée pour cette étude. Dans ce schéma d'attaque, nous considérons que les hôtes cibles se situent dans le réseau interne, protégé par la solution de sécurité. Les machines attaquantes se situent à l'extérieur du réseau et doivent transiter par la machine intermédiaire pour mener leurs attaques.

Au travers de cette étude, nous avons choisi d'observer le comportement de Snort et d'un firewall commercial. Snort [76], en plus d'être un IDS open-source, se repose sur une large communauté qui enrichit sa base de signatures, ce qui fait d'elle un élément de choix pour mener des expérimentations. Nous l'avons comparé à un firewall industriel, dont nous ne donnerons pas excessivement de détail à la demande de son propriétaire. Ces deux solutions de sécurité sont configurées par défaut et le trafic réseau est uniquement généré par l'attaque en cours. L'attaque de balayage de ports est relativement simple à réaliser et un IDS est plutôt efficace dans un contexte habituel, ce qui nous permet de mettre en lumière la faiblesse de ces derniers dans un

2. Il existe toutefois des moyens pour arriver aux mêmes conclusions : le re-jeu, l'analyse des réponses, etc.

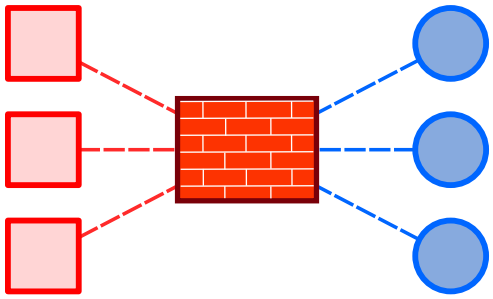


FIGURE 3.2 – Schéma du banc d'essai avec un point d'entrée

contexte distribué.

Ces expérimentations, réalisées dans des environnements différents (différents nombres d'attaquants, de cibles, d'éléments de sécurité, de types de balayages de ports ou encore de manières de distribuer l'attaque) indiquent que les solutions existantes ne sont pas adaptées pour détecter les attaques distribuées. Pour quantifier l'efficacité de ces attaques, nous calculons le succès de l'attaque, exprimé par la formule suivante :

$$\text{succès} = \frac{\text{nombre de balayage de ports réalisés sans être détecté}}{\text{nombre de balayage de ports total}}$$

Les résultats des expérimentations, dont les aspects importants sont repris, démontrent que les solutions de sécurité ne détectent pas efficacement des attaques dès lors qu'elles sont distribuées. En effet, dans le cadre de balayage de ports légitime et pour la majorité des environnements de test, 32 hôtes attaquants sont suffisants pour conduire une attaque sans se faire détecter. Nous illustrons ces résultats au travers de la Figure 3.3. Celle-ci représente le taux d'efficacité d'une attaque distribuée ciblant 4 hôtes en fonction du nombre d'attaquants. Une valeur de 100% correspond à une attaque distribuée menée complètement, sans détection de la part de la solution de sécurité. Dans cet environnement de test, seulement 8 machines attaquantes sont suffisantes pour ne pas être détecté.

On peut remarquer que, pour les deux IDS considérés, augmenter le nombre de machines attaquantes améliore significativement le succès de l'attaque. Ceci s'explique simplement en étudiant les méthodes de détection utilisées par ces systèmes. Bien que les IDS soient efficaces lors de balayages de ports illégitimes (car ces attaques peuvent être représentées au travers de signatures), ils ne peuvent pas conclure qu'un hôte est effectivement en train de mener une attaque dans le cas d'un balayage de ports légitime. Chaque machine attaquante ne réalise qu'une sous-partie de l'attaque et dans sa vision globale, l'IDS ne peut pas convenir qu'une attaque est en cours. De nos jours, obtenir de telles ressources est ridiculement facile, grâce à des botnets, après une infection virale ou tout simplement en louant des ressources auprès d'un Cloud Computing.

La structure du banc d'essai étant assez éloignée de structures réseaux réelles, nous proposons dans la prochaine section des expérimentations qui se concentrent sur l'impact des attaques distribuées sur différentes topologies réseau.

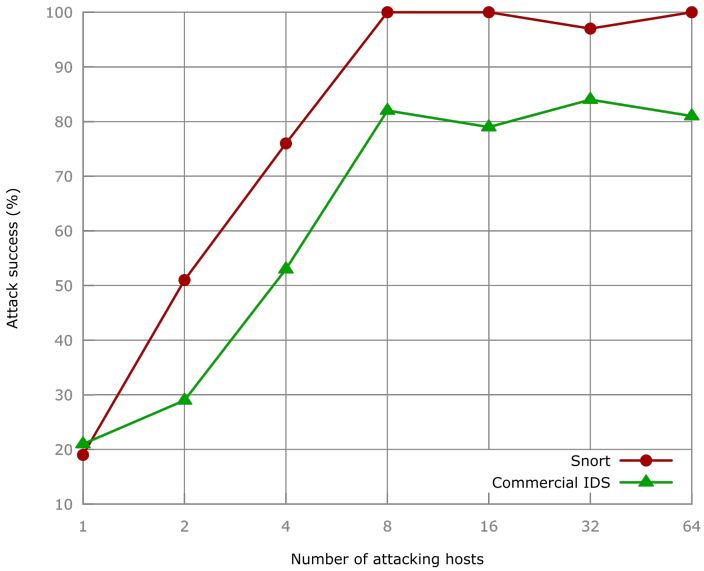


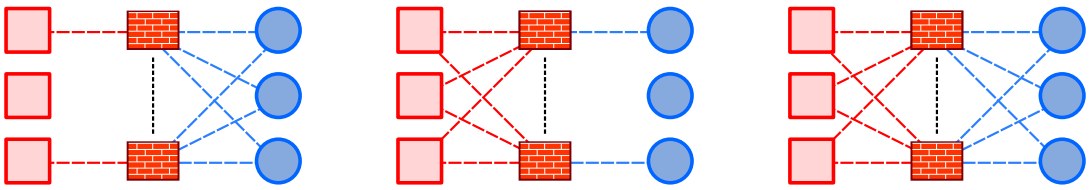
FIGURE 3.3 – Succès de l'attaque distribuée sur 4 cibles en fonction du nombre d'attaquants

3.2.4 Impact de la topologie réseau

La première série d'expérimentations est un cas de figure favorable à la solution de sécurité, parce qu'elle peut analyser l'intégralité du trafic de l'attaque. La démultiplication des attaquants arrive toutefois à déjouer les mécanismes de détection, en diminuant l'attention individuelle de chacun des hôtes qui prend part à l'attaque. La topologie étudiée, où une seule solution de sécurité est le point d'entrée du réseau à attaquer, est assez éloignée des structures déployées réellement. En effet, pour des structures comme le Cloud Computing, plusieurs points d'entrée existent notamment pour pouvoir assurer des propriétés comme la haute disponibilité.

Dans le cas où plusieurs chemins existent pour atteindre l'ensemble des machines cibles de l'attaque, la situation empire du point de vue de la sécurité globale du réseau. En effet, chaque solution ne voit qu'une sous partie de l'attaque et la qualité de la sécurité est dégradée. C'est ce que nous avons cherché à vérifier aux travers de notre seconde série d'expérimentations [74]. Nous avons réalisé les mêmes expérimentations, mais nous avons considéré les topologies illustrées dans la Figure 3.4.

Cette figure illustre une structure réseau où deux solutions de sécurité jouent le rôle de point d'entrée. Nous avons distingué trois déclinaisons de topologies. La topologie T1 correspond à une isolation géographique des nœuds attaquants. Le trafic d'un nœud attaquant sera donc analysé par un seul nœud d'analyse, mais plusieurs attaquants peuvent utiliser différents points d'entrée pour s'adresser à la même machine interne. La topologie T2 propose l'inverse : une isolation des noeuds cibles de l'attaque. Dans ce cas, les solutions de sécurité surveillent intégralement le trafic dirigé vers une cible. La topologie T3 propose une interconnexion totale entre les attaquants et les cibles de l'expérimentation. Dans ce cas, un nœud attaquant peut utiliser plusieurs chemins



(a) Topologie T1 (b) Topologie T2 (c) Topologie T3

FIGURE 3.4 – Topologies étudiées dans le cas d'un environnement à deux routes

pour atteindre une cible.

Les mesures ont été réalisées sur la plateforme académique Grid'5000, permettant de déployer un grand nombre d'attaquants, de cibles et de solutions de sécurité. Nous avons réalisé les mêmes expérimentations que celles décrites dans la Section 3.2.3, tout en modifiant les topologies réseaux et le nombre de solutions de sécurité. Introduire de multiples points d'entrée revient donc, dans notre cas, à obtenir des attaques multi-chemins. Nos résultats sont quantifiés en mesurant le gain d'une attaque, qui consiste à comparer le succès d'une attaque multi-chemins au succès d'une attaque sur un seul chemin. Par exemple, un gain de 2 équivaut à une attaque qui est deux fois plus efficace par rapport à une structure réseau où un seul point d'entrée existe.

La Figure 3.5 illustre les résultats obtenus durant cette étude. Cette courbe représente le gain d'une attaque multi-chemin en fonction du nombre de solutions de sécurité intermédiaires. Ces résultats sont extraits plus spécifiquement d'un environnement qui implique 64 hôtes cibles, 32 attaquants et la solution déployée est Snort.

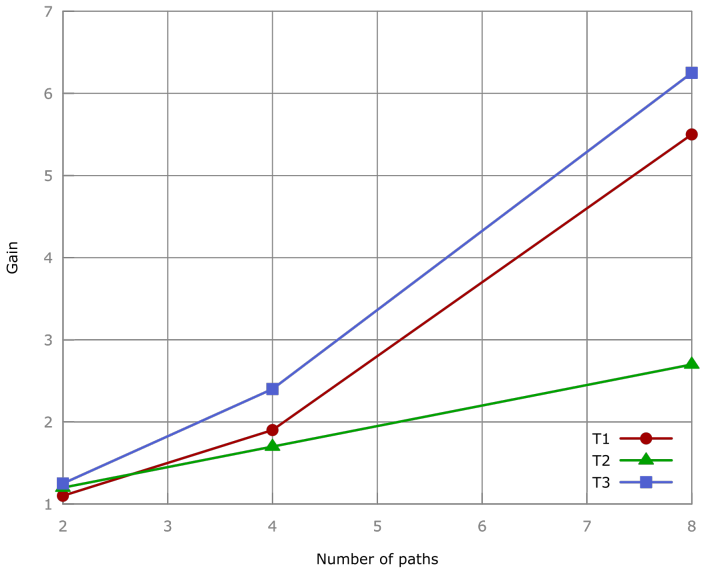


FIGURE 3.5 – Gain obtenu sur le succès de l'attaque en fonction du nombre de chemins

Ces résultats ont plusieurs aspects intéressants. Premièrement, les topologies T1 et T3 sont favorables aux attaquants, pratiquement proportionnellement au nombre de routes possibles. Dans ces cas de figures, les solutions de sécurité ne voient qu'une sous partie du trafic généré vers une machine cible. D'autre part, la topologie T2 n'offre pas un gain aussi intéressant que les deux autres topologies. Les solutions de sécurité sont plus efficaces dans cet environnement, grâce aux mécanismes de détection utilisés. Dans cette topologie, une solution de sécurité voit l'intégralité du trafic vers une machine cible, de la même manière que s'il n'y avait qu'un seul chemin vers cette machine.

En s'appuyant sur nos explications quant aux méthodes de détection des balayages de ports, nous savons qu'une technique couramment utilisée consiste à dénombrer les événements suspicieux. Pour cette technique, le nombre d'événements suspicieux est associé soit à la machine attaquante, soit à la machine cible. Une fois que ce nombre dépasse un seuil fixé, le système de sécurité en conclut qu'une attaque est en cours et une alerte est générée. Dans le cas d'attaques distribuées, l'association de ce nombre à la machine attaquante n'est pas probante, puisque cette machine en question ne réalisera qu'une partie substantielle de l'attaque. Par contre, l'association du nombre d'événements suspicieux à la machine cible est plus efficace, comme nous le dénombre les résultats de la topologie T2.

Ces mesures montrent que dans le cadre de structures multi-chemins, c'est-à-dire où il y a plusieurs solutions de sécurité faisant office de points d'entrée, les attaques distribuées se révèlent efficaces. En plus de la distribution de l'attaque, le fait qu'une solution de sécurité ne voit qu'une sous partie du trafic malveillant minimise la détection de l'attaque. Pour palier à ce problème, il est possible d'utiliser des solutions capables de collaborer et de prendre des décisions en rapport avec l'attaque vue dans sa globalité. C'est sur ce constat que nous avons élaboré Discus, présenté dans la prochaine section.

3.3 Discus : une nouvelle architecture de solutions de sécurité

Dans la section précédente, nous avons présenté nos expérimentations, tout d'abord en nous concentrant sur l'efficacité des attaques distribuées, puis sur l'aspect multi-chemins des structures telles que le Cloud Computing. Nous montrons au travers de ces mesures que les attaques distribuées sont efficaces, particulièrement dans un contexte multi-chemins. En effet, bien que plusieurs solutions de sécurité soient déployées dans le réseau, cela ne suffit pas à détecter les attaques. Cela démontre qu'il n'est pas suffisant de démultiplier le nombre de noeuds d'analyse sur le réseau, mais qu'il est nécessaire de corréler un maximum d'informations susceptibles de permettre une détection d'attaque. C'est ce que nous proposons au travers de notre solution, Discus, que nous présentons dans cette section.

Dans un premier temps, nous présenterons l'architecture de Discus et son fonctionnement global. Ensuite, nous nous concentrerons sur les acteurs de cette solutions, les sondes de sécurité. Pour finir, nous discuterons des verrous scientifiques que nous visons au travers de nos travaux.

3.3.1 Présentation de l'architecture DISCUS

Nos travaux s'axent autour de DISCUS, l'architecture que nous proposons pour palier aux problèmes évoqués auparavant. La culture de notre équipe de recherche nous a orienté vers une approche « *système embarqué* », où nous cherchons à installer un maximum de solutions de sécurité au plus proche des hôtes à surveiller. Notre architecture, qui se repose sur le concept des IDS collaboratifs, a été élaborée en vue d'être déployée sur des structures complexes, telles que le Cloud Computing.

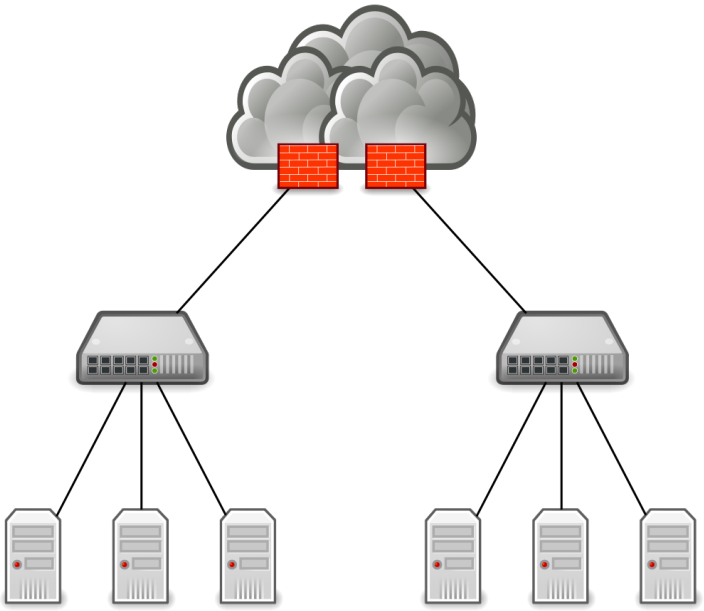


FIGURE 3.6 – Topologie réseau ciblée par DISCUS

De manière schématique, DISCUS s'intéresse aux topologies réseau illustrées dans la Figure 3.6. Dans cette topologie, les firewalls sont les composants réseau qui font l'intermédiaire avec le réseau externe. Ils peuvent être multiples et opèrent de manière habituelle. Ensuite, les composants réseau habituels, tels que les commutateurs, forment un certain nombre de couches réseau. Pour finir, les machines physiques et les machines virtuelles sont situées dans le bas de cette figure. Notre solution cherche à surveiller et protéger ce type de structures, mais comme nous l'avons remarqué au travers des expérimentations, les solutions usuelles ne sont plus suffisantes, même lorsqu'elles sont nombreuses. Un degré suffisant de collaboration est nécessaire pour détecter les attaques à large échelle. En ce qui concerne les attaques internes au réseau, il est nécessaire qu'au moins un nœud de sécurité soit présent sur le chemin réseau de l'attaque.

C'est pourquoi nous avons ajouté deux types de sondes réseau : les sondes physiques et les sondes virtuelles. Ces sondes sont insérées dans le réseau et placées au plus proche des machines à surveiller. Les sondes virtuelles sont logiciellement intégrées sur les machines physiques ou virtuelles pour analyser les événements locaux et participer à la sécurité globale du réseau. Quand

aux sondes physiques, il s'agit d'équipements physiques placés en coupure sur le réseau. Il est même concevable de les intégrer directement aux commutateurs, équipés de puissants composants physiques (FPGA). L'ensemble de ces sondes forment un réseau massivement distribué de sondes de sécurité. Cependant, nous ne nous satisfaisons pas des sondes pour former notre CIDS. En effet, les éléments de sécurité usuels font partie de la structure : les gros firewalls de cœur et les IDS installés par les clients sont à même de fournir des informations pertinentes.

Nous sommes donc confrontés à plusieurs cibles distinctes : les sondes physiques, les sondes virtuelles et les solutions traditionnelles de sécurité. De plus, parmi les sondes physiques, on peut distinguer les sondes dont la programmation est logicielle de celles dont la programmation est orientée électronique. Développer un logiciel pour l'ensemble de ces cibles n'est pas aisé car cela requiert une expertise dans différents domaines de programmation. Il est donc nécessaire de trouver un moyen pour qu'un utilisateur puisse déployer facilement une politique de sécurité sur ces cibles, sans pour autant être un expert dans tous les domaines précédemment cités. D'autre part, il faut remarquer qu'il existe un panel de profils d'utilisateurs assez varié. En effet, comme nous l'avons mentionné dans la première partie de cette problématique, l'expertise et l'intention des utilisateurs sont différents suivant l'utilisateur en question. Par exemple, le développeur de solution de sécurité écrit une politique de sécurité généraliste, alors que l'administrateur en sécurité souhaite spécifier la politique en fonction de l'architecture à surveiller. Ces différents profils amplifient davantage le besoin d'abstraction, qui permettrait à chacun de ces utilisateurs d'intervenir dans l'élaboration de la politique de sécurité.

Nous avons présenté succinctement Discus dans cette section, en détaillant les différents composants de l'architecture. Nous avons également mis en lumière les problèmes que nous adressons avec nos travaux. Les verrous scientifiques que nous souhaitons adresser au travers de Discus sont explicités dans la prochaine section, où nous rentrons plus dans le détail du fonctionnement de notre solution.

3.3.2 Les verrous scientifiques soulevés par DISCUS

Dans la première section de ce chapitre, nous soulevons les différentes problématiques liées au Cloud Computing, telles que la configuration des solutions de sécurité ou la volatilité des services déployés. Nos travaux s'intéressent à deux axes majeurs : la configuration et le déploiement des sondes réseau, et la collaboration de celles-ci. Nous détaillons ces deux problématiques dans les prochaines sous sections, en expliquant comment Discus peut palier à ces problèmes.

Configuration et déploiement des sondes

Notre architecture est composée de nombreuses cibles, qui présentent chacune des particularités. Comme nous souhaitons que ces cibles soient configurables par un grand nombre d'utilisateurs, quelque soit son expertise, il est nécessaire de trouver un moyen uniforme pour configurer ces sondes.

L'utilisation d'un langage dédié est une solution idéale permettant de gérer l'hétérogénéité des cibles. En effet, la représentation dans un tel langage est agnostique des spécificités des cibles. Le langage offre une couche d'abstraction qui permet de ne plus se concentrer sur la programmation système de la sonde, mais plutôt sur l'écriture de la politique de sécurité. Ce langage permet également une plus forte synergie dans l'écriture de cette politique : de nombreux acteurs peuvent prendre part à son élaboration, qu'il s'agisse de l'écriture de règles de sécurité générales ou bien spécifiques à l'infrastructure surveillée. Pour finir, l'utilisation d'un tel langage permet également d'obtenir une représentation de la politique qui peut être étudiée en vue d'optimisations.

Pour configurer l'architecture de sécurité, nous proposons DISCUS SCRIPT, un langage dédié à la détection d'intrusions. Ce langage permet à un expert de la sécurité réseau de décrire de manière événementielle ce qu'il souhaite détecter. C'est également en utilisant ce langage qu'il décrira le comportement collaboratif de notre solution, précisé dans la prochaine section.

L'utilisation de ce langage est schématisée dans la Figure 3.7. Tout d'abord, l'utilisateur écrit un ensemble de règles grâce à DISCUS SCRIPT, sans pour autant prendre en considération les cibles sur lesquelles ces règles seront déployées. Ensuite, il utilise la chaîne de compilation pour générer les fichiers spécifiques à chacune des cibles. La génération de ces fichiers nécessite de connaître l'architecture du réseau à protéger et les informations relatives aux sondes déployées. Pour finir, l'utilisateur déploie les solutions.

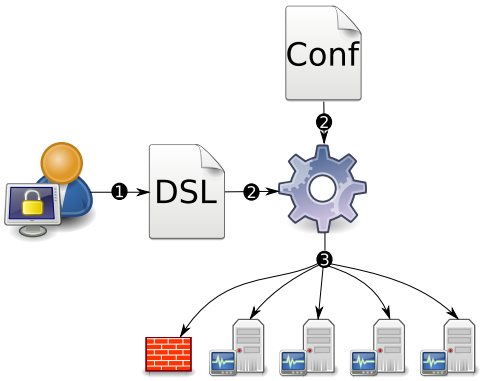


FIGURE 3.7 – Utilisation du langage dédié

Distribution et Collaboration

L'élaboration de DISCUS s'intéresse à un autre verrou majeur : la distribution de solutions de sécurité et la détection collaborative d'intrusions. Nous détaillons dans les prochains paragraphes plusieurs axes de recherche.

Tout d'abord, il est nécessaire d'identifier une structuration adaptée aux infrastructures larges telles que le Cloud Computing. Notamment, l'architecture de ces réseaux peut être complexe, comme nous l'illustrons dans la Figure 3.8. Dans ce schéma, plusieurs infrastructures locales sont connectées entre elles en traversant internet. Nos travaux s'intéressent autant aux infrastructures

locales qu'à la structure en sa globalité. Il est donc légitime de se questionner quant à la nature des liens existants entre l'ensemble des solutions de sécurité. Notamment, nous pouvons nous poser la question suivante : « *est-il préférable d'adopter une approche centralisée, hiérarchique ou complètement distribuée ?* ». Jusqu'à présent, nos travaux s'orientent vers une architecture hybride, qui se basera sur la topologie du réseau à protéger. Il est toutefois nécessaire d'évaluer quelle structuration est la plus efficace.

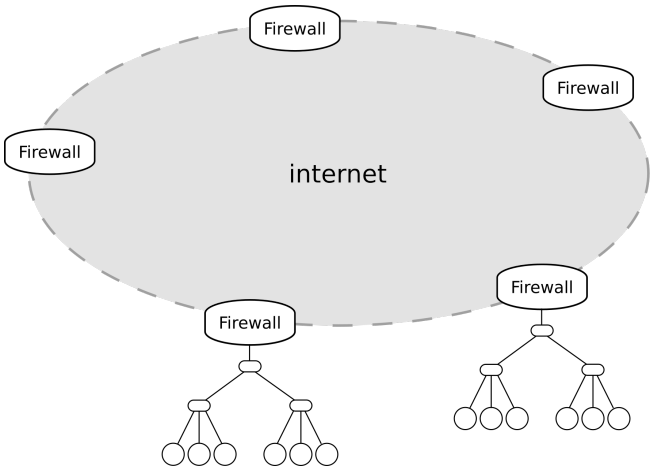


FIGURE 3.8 – Exemple d'infrastructure réseau complexe

D'autre part, la collaboration s'effectue au travers de communications entre les sondes du réseau. Il est donc nécessaire de s'intéresser à la nature des échanges et d'établir un moyen de communication que chacune des solutions peut adopter. En ce qui concerne la détection collaborative d'intrusions, nous avons introduit dans notre langage le concept de *tables*, assimilables aux structures de données en C. Ces structures de données sont partagées au sein du réseau et chacun des nœuds peut consulter et manipuler ces tables. Ce concept soulève de nombreuses problématiques, telles que :

— le placement des tables sur le réseau, autrement dit attribuer à chacun des nœuds du réseau une sous-partie des données à héberger localement ;

— la nature des communications pour manipuler les tables et éventuellement la mise en place de mécanismes de réplication ou de cache des données ;

— les mécanismes de tolérance aux fautes et la possibilité de supporter des pannes d'un des nœuds du réseau.

3.3.3 Synthèse

Dans cette section, nous avons présenté DISCUS, une solution qui s'intéresse à la sécurité d'infrastructures réseau. Elle est constituée d'un réseau massivement distribué de sondes qui participent à la sécurisation du réseau en sa globalité en collaborant entre elles. Pour proposer

une analyse du trafic réseau précise, il nous est nécessaire de placer ces sondes au plus proche des machines à surveiller. C'est pourquoi nous disposons d'un parc de sondes hétérogènes, qui peuvent être placées physiquement entre les équipements réseau , ou bien installées virtuellement lorsque nous désirons surveiller les réseaux virtuels.

Cette solution soulève de nombreuses problématiques que nous avons regroupé en deux axes. Le premier s'intéresse à la configuration, la mise à jour et le déploiement des sondes sur un parc important. Il n'est pas envisageable de concevoir qu'une personne réalise ces tâches manuellement, c'est pourquoi nous proposons un langage dédié. DISCUS SCRIPT est un langage qui permet de se concentrer sur le développement d'une politique de sécurité, notamment en s'appuyant sur une couche d'abstraction réseau et matérielle. C'est ensuite à la charge de la chaîne de compilation de générer les fichiers nécessaires au déploiement de chacune des sondes.

Le second axe concerne la distribution et la collaboration de ces sondes. Il est nécessaire d'établir un moyen de communication entre l'ensemble de ces sondes, c'est pourquoi il est indispensable d'évaluer quelle structure est la plus adaptée, ainsi que la nature des messages à échanger et le rôle de chacune des sondes. Nous introduisons dans le langage DISCUS SCRIPT la notion de tables distribuées, assimilable à une base de données distribuée. Connaissant la topologie réseau et les capacités de chacune des sondes, il faut déterminer où placer les données, ainsi que les mécanismes mis en place, tels que la tolérance aux pannes ou la détection de sondes byzantines.

Nos travaux s'orientent principalement sur le premier axe, c'est-à-dire la proposition d'un langage dédié décrivant une politique de sécurité, qui nous permettra ensuite de configurer les sondes de la solution distribuée. Nous proposons toutefois la notion de tables, qui nous permettra dans de futurs travaux de distribuer les informations entre plusieurs sondes et de pouvoir donc collaborativement détecter les intrusions. Pour finir, le Chapitre 5 s'intéresse à l'évaluation de notre langage.

4

Discus Script : un langage dédié à la sécurité réseau

Nos travaux se concentrent sur la détection d'intrusions et d'attaques sur un réseau comportant de nombreuses machines, tel que le Cloud Computing. Dans le chapitre précédent, nous avons proposé DISCUS pour répondre aux problématiques associées à la sécurité de ces larges réseaux. Parmi les verrous technologiques et scientifiques que nous avons identifiés, nous avons évoqué l'hétérogénéité des solutions de sécurité, leur configuration et leur déploiement. D'autre part, notre approche souhaite faire intervenir un grand nombre d'acteurs : le développeur de solution de sécurité, l'expert en sécurité étudiant les nouvelles failles, le client ou encore les administrateurs sécurité et réseau d'une architecture à protéger. Il serait alors nécessaire que chacun de ces acteurs ait un certain degré d'expertise dans le développement et le déploiement de chacun des types de nœuds à déployer, ce qui n'est pas envisageable dans une vraie structure réseau. Notre langage DISCUS SCRIPT émane du besoin d'un socle commun de communication entre ces acteurs. Il propose également une couche abstraite de développement, qui permet de se détacher des spécificités techniques des nœuds à déployer.

Nous présentons dans ce chapitre DISCUS SCRIPT, ses particularités et son usage. Dans un premier temps, la Section 4.1 introduit le langage, ses objectifs et ses particularités. Ensuite, nous proposons une présentation du langage, de sa syntaxe et sa grammaire dans la Section 4.2. Pour finir, les Sections 4.3 et 4.4 présentent des cas d'utilisations du langage.

4.1 Présentation de DISCUS SCRIPT

L'élaboration d'un langage dédié n'est pas une chose aisée, car il est nécessaire d'identifier les concepts du domaine d'étude ciblé et de proposer des moyens simples pour les exprimer. Dans cette section, nous détaillons les caractéristiques générales du langage et expliquons les différents choix que nous avons réalisés lors de la création de DISCUS SCRIPT.

4.1.1 Motivations

Comme nous l'avons déjà évoqué, notre choix de reposer DISCUS sur un langage dédié est motivé par deux axes principaux : le grand nombre d'acteurs impliqués dans l'écriture de la politique de sécurité, et l'hétérogénéité des cibles. Un langage dédié permet d'exprimer plus facilement des concepts propres au domaine d'étude visé et offre la possibilité d'en extraire des propriétés facilitant l'optimisation des règles écrites dans ce langage.

De nombreux langages existent dans la littérature, tels que Snort, Bro ou Lambda. Nous nous sommes tournés vers la création d'un nouveau langage car nous ne trouvions pas les propriétés souhaitées dans ces langages. Notamment, nous souhaitions un langage qui permet :

— de décrire des attaques comme une séquence d'événements, à la granularité du paquet réseau ;

— de s'abstraire de la cible sur laquelle sera déployée la politique de sécurité ;

— d'écrire des règles de manière intuitive, quelque soit le niveau d'expertise de l'utilisateur [1] ;
— de décrire un comportement collaboratif entre les sondes du réseau ;
— de détecter les incohérences dans la politique de sécurité, de les indiquer à l'utilisateur et de lui proposer des corrections.

Ne trouvant pas l'ensemble de ces propriétés dans les langages existants de la littérature, nous avons décidé de proposer Discus Script. Notre langage, accompagné de la chaîne de compilation, permet d'exprimer une politique de sécurité puis d'en générer un fichier spécifique (binaire ou source) pour chaque sonde du réseau sécurisé. Nous présentons dans les prochaines sections les caractéristiques du langage.

4.1.2 Programmation événementielle

Notre langage dédié, Discus Script, propose d'écrire une politique de sécurité au travers d'une suite de règles. Nous avons orienté l'écriture de ces règles de manière événementielle, c'est-à-dire qu'une règle réagira à un événement donné. L'événement élémentaire que nous ciblons est le paquet réseau.

Nous avons choisi cette approche événementielle car l'analyse de paquets est intrinsèquement événementielle. En effet, l'analyse de détection d'intrusions débute à la réception d'un paquet. D'autre part, le déclenchement d'un événement n'est pas uniquement réalisé lors de la réception d'un paquet, mais peut être initié par une règle qui souhaite affiner une analyse. Dans le cas de Discus Script, certaines règles ne seront déclenchées que dans certaines conditions, ce qui permet de minimiser la durée d'analyse d'un paquet.

D'autre part, un autre avantage de cette approche événementielle est de pouvoir tirer profit des architectures naturellement parallèles. Qu'il s'agisse des architectures multi-cœurs ou des circuits électroniques tels que les FPGA, l'analyse d'un ou plusieurs paquets peut se dérouler de manière parallèle.

Cela donne la possibilité de représenter les règles de sécurité sous forme de graphe d'événements, dont la racine est l'événement primaire `packet`. Cette représentation permet de réaliser des analyses et conclure que certains événements peuvent être supprimés. Nous détaillerons les manipulations possibles dans les prochaines sections. La Figure 4.1 représente un exemple de graphe issu d'un fichier source Discus Script.

4.1.3 Discus Script, un langage statiquement typé

Un des objectifs fixés dans la section précédente concerne la capacité du langage à détecter des incohérences dans le fichier source et de pouvoir le signaler à l'utilisateur. Pour atteindre cet objectif, nous avons mis plusieurs mécanismes en place, dont le fait que le langage soit

1. en considérant à minima qu'il ait des notions en réseau et sécurité

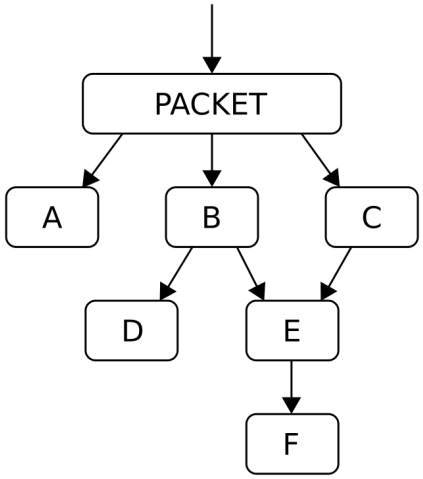


FIGURE 4.1 – Exemple de graphe d'événements

statiquement typé. Cela permet d'établir une phase de vérification stricte au moment de la compilation pour vérifier la compatibilité des types employés dans les différentes expressions du langage. Nous présentons les types de base dans les prochains paragraphes.

intN est le type représentant les entiers dont la taille, exprimée en nombre de bits, est toujours connue. Ainsi, il sera possible d'utiliser les types `int1`, `int2`, ..., `intN` pour des entiers de respectivement 1, 2, ..., `N` bits. Les expressions et variables de ce type peuvent être sujettes aux opérations usuelles sur les entiers, telles que les opérations arithmétiques, logiques ou comparatives. Le type de retour de ces différents opérateurs est susceptible d'avoir une taille différente (en nombre de bits) que celui des opérandes. Par exemple, une addition entre deux entiers produira un entier possiblement plus grand (en nombre de bits) que les deux opérandes. À l'inverse, des opérations logiques génèreront des entiers de type `int1`, assimilables à des booléens.

float permet de représenter des nombres réels. Contrairement aux entiers, la taille des nombres réels est fixée. En fonction des cibles, la chaîne de compilation de DISCUS SCRIPT produira du code spécifique, par exemple en réalisant du calcul en virgule fixe à l'aide de nombres entiers, ou bien en utilisant un circuit dédié.

enum permet d'utiliser les types énumérés. Les variables d'un tel type ne pourront prendre qu'un ensemble de valeurs connues à l'avance. Nous avons choisi d'introduire ce type dans DISCUS SCRIPT pour faciliter la description des protocoles réseau, pour lesquels l'objet d'un paquet réseau réside dans un champ spécifique de l'en-tête. Par exemple, le champ `OPER` du protocole ARP (qui permet d'associer une adresse MAC à une adresse de la couche réseau) permet d'identifier le type de l'opération : requête ou réponse. Nous pourrions alors écrire dans notre langage le type

énuméré tel que décrit dans le Listing 4.1. L'écriture de ces énumérations facilite également la description des protocoles réseau, où il est fréquent de décrire l'état actuel d'une connexion grâce à un automate à états.

Listing 4.1– Exemple d'utilisation d'un type énuméré

```
1 enum arp_oper_values {arp_request=1, arp_reply};
```

time correspond au type représentant une estampille temporelle. Le mot clé du langage `now` permet de connaître l'heure actuelle. La chaîne de compilation permet de générer une estampille dont la précision dépend de la cible. Une expression de type `time` est manipulable de la même manière qu'un entier.

bitstream représente une séquence de bits dont la taille n'est pas connue. Idéal pour la représentation de paquets réseau, le type `bitstream` est utilisé pour l'événement élémentaire `packet`. En considérant que la variable `a` est de type `bitstream`, le langage fournit un ensemble d'opérations sur les flux de bits comme par exemple :

— `a[0:N]`, qui permet d'extraire un entier de N+1 bits ;

— `a[92:]`, qui permet d'extraire une sous partie du flux, à partir du 92ème bit ;

— `a + b`, qui permet de concaténer les bitstreams `a` et `b` ;

— `"HTTP" in a`, qui permet de chercher le motif `HTTP` dans le bitstream `a`.

ipaddr représente une adresse de machine ou de réseau (IPv4 ou IPv6). Plusieurs opérations du langage permettent de manipuler ces adresses, comme par exemple :

— `a in b`, qui permet de tester si l'adresse `a` appartient au réseau `b` ;

— `net4(a)` et `net6(a)` qui permet de construire respectivement une adresse IPv4 et IPv6 depuis le bitstream `a`.

4.1.4 Aspect collaboratif via le mécanisme des tables

Un des objectifs de nos travaux est de proposer une architecture qui permette la collaboration des nœuds de sécurité. Le mécanisme des tables est au cœur de cette collaboration. Emprunté au vocabulaire des systèmes de gestion de bases de données, les tables permettent de stocker des informations relatives à l'état du réseau et peuvent être consultées ou modifiées par les sondes. À l'instar des bases de données, nos tables sont composées d'un ensemble d'enregistrements qui respectent une structure, constituée de champs typés par un type de base du langage.

En fonction de la topologie réseau, de la spécificité de chacune des sondes, la chaîne de compilation va décider de la stratégie à adopter en ce qui concerne la distribution des données. Un de nos souhaits étant de simplifier la tâche de développement de la politique de sécurité, notre chaîne de compilation génère automatiquement l'algorithme adopté, en fonction de la topologie. De cette manière, le développeur ne se préoccupe pas de l'approche utilisée et peut se concentrer sur les règles de sécurité. Nos travaux ne s'intéressant pas spécifiquement à cet aspect, notre approche, basée sur la réplication, reste assez simple et donc assez inefficace.

Listing 4.2– Exemple de déclaration d'une table

```
1 table active_connections {
2     ipaddr local_host;
3     ipaddr remote_host;
4     time timestamp;
5 };
```

Le Listing 4.2 illustre une déclaration de table. Cette table permet de retenir les informations relatives à une connexion TCP, à savoir l'adresse locale, l'adresse distante et l'heure d'établissement de connexion. Cet exemple reste assez sommaire, c'est pourquoi le lecteur trouvera d'autres exemples plus complets dans la prochaine section. Les opérations sur les tables, telles que les insertions, les mises à jour ou les consultations sont les principales actions que pourront effectuer les règles de sécurité.

4.1.5 Chaîne de compilation

Une fois la conception d'un langage terminée, un certain nombre d'outils est développé pour manipuler les fichiers sources et les interpréter. Nous appelons cet ensemble d'outils une chaîne de compilation. Nous présentons dans la Figure 4.2 un schéma représentant l'ensemble des outils présents dans la chaîne de compilation de DISCUS SCRIPT.

Tout d'abord, le compilateur se charge de la lecture des fichiers sources DISCUS SCRIPT. Il est composé de plusieurs outils, tels que l'analyseur lexical, grammatical ou sémantique. Leur rôle est de s'assurer que les mots du langage, appelés unités lexicales ou *tokens* en anglais, forment des phrases grammaticalement correctes et qui ont du sens. Cette première phase produit un fichier intermédiaire, correspondant à un fichier source auquel des informations contextuelles ont été ajoutées.

Plusieurs outils peuvent manipuler ces fichiers intermédiaires, tels que :

— les outils d'analyse, qui se chargent de détecter les informations incohérentes et signalent à l'utilisateur les modifications à réaliser ;

— les outils d'optimisation, qui améliorent la représentation intermédiaire du fichier source, grâce à des analyses préalables, afin d'optimiser le fichier intermédiaire ou les performances du fichier généré ;

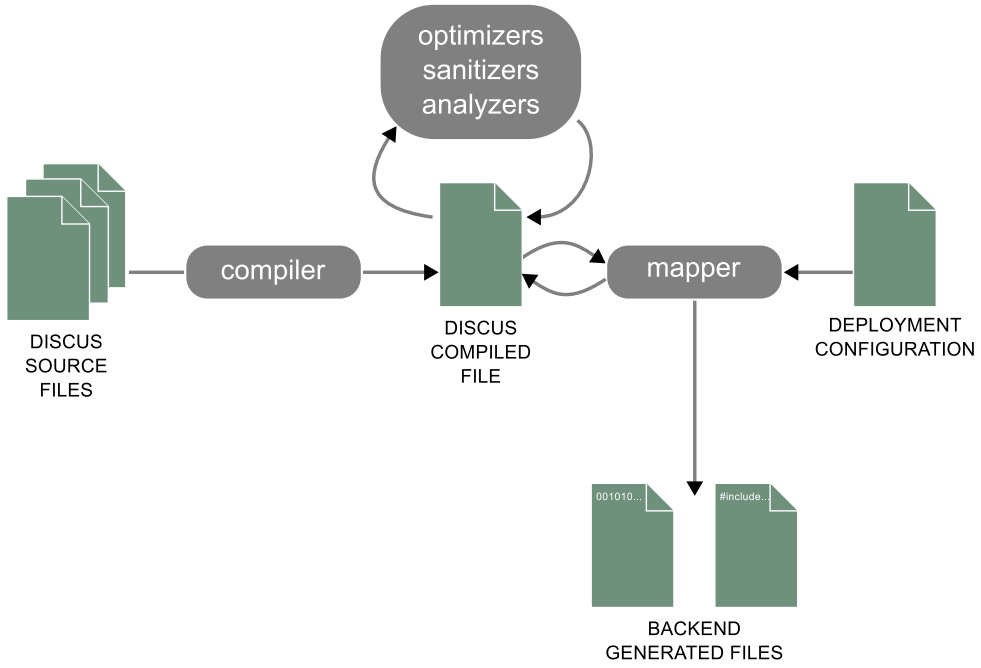


Figure 4.2 – Chaîne de compilation de Discus Script

— les outils de nettoyage, qui se chargent d'enlever certaines parties d'un fichier intermédiaire, car définies comme inutiles par les deux types d'outils précédents.

Les outils proposés dans notre chaîne de compilation se chargent d'analyser le contenu d'un fichier source et de déterminer si des incohérences sont présentes. Par exemple, nous avons des outils capables de détecter si des cycles sont présents dans le graphes d'événements, que tous les événements sont atteignables ou encore d'éliminer les branches d'événements qui mènent à un événement feuille qui ne produit aucune action.

Pour finir, l'outil de *mapping* génère les fichiers de configuration des sondes. Il peut s'agir de fichiers sources, de fichiers binaires ou encore de fichiers de configuration. À ce jour, l'outil génère des fichiers en C. Cet outil s'appuie sur un fichier de représentation de la topologie pour générer des fichiers spécifiques à chaque cible. Connaissant les spécificités d'une sonde (par exemple sa capacité mémoire) et son placement sur le réseau, l'outil peut par exemple manipuler le fichier de représentation intermédiaire pour supprimer les événements inutiles, car non applicables dans cette configuration réseau.

4.1.6 Propriétés du langage

Dans cette partie, nous mettons en lumière des propriétés du langage dédié, obtenues grâce à sa construction ou l'utilisation des outils associés au langage.

Langage extensible

Lors de l'élaboration de DISCUS SCRIPT, nous nous sommes orientés vers un compilateur généraliste, c'est-à-dire qui n'a pas de préjugés sur la sonde ciblée ou le support réseau. Autrement dit, nous voulions qu'aucun élément spécifique à l'analyse réseau ou aux protocoles réseaux ne soit contenu dans le compilateur ou la chaîne de compilation. Nous entendons par là que l'utilisation seule du langage est suffisante pour décrire les différentes couches réseau analysées. Cela constitue également une preuve que notre langage est expressif et utilisable dans le contexte de description de protocoles. En effet, des fichiers (assimilables à des libraires) écrits en DISCUS SCRIPT décrivent les protocoles réseaux tels qu'Ethernet, IPv4 ou TCP. Cette approche permet également de favoriser une dynamique communautaire, où des individus viennent enrichir les fichiers de base, qui contiennent des règles généralistes sur les protocoles réseaux.

L'événement élémentaire `packet` est spécifique à chacune des sondes et nos développements actuels considèrent qu'il s'agit du protocole Ethernet. Mais avec cette approche, il est envisageable de cibler d'autres événements élémentaires, tels que des paquets Wifi par exemple. Cependant, notre approche n'est pour le moment pas transposable dans le cas de communication en flux.

Détection de règles incohérentes ou non pertinentes

Nous signalons dans le précédent paragraphe que notre langage encourage l'aspect communautaire, où des experts écrivent des règles généralistes et enrichissent un ensemble de librairies issues de la communauté. Également, un des objectifs de notre langage est de faire intervenir un grand nombre d'acteurs sur l'élaboration de ces fichiers. Pour finir, le même fichier source sera utilisé, puis raffiné, pour générer des fichiers spécifiques à chaque sonde réseau.

De ces choix résulteront des fichiers de règles riches, dont certaines seront incohérentes ou non pertinentes. Par exemple, des règles peuvent ne pas être nécessaires pour certaines configurations réseau, ou tout simplement mener à des règles *feuilles* [2] qui ne font aucune action. Nous considérons ici qu'une action est une manipulation d'une table ou la levée d'une alerte. Dans ce cas, il est possible d'optimiser le graphe d'événements pour avoir une politique de sécurité spécialisée pour l'environnement.

La Figure 4.3 illustre un graphe qui peut être optimisé. En effet, plusieurs événements peuvent être supprimés de cette représentation. Dans ce schéma, un événement qui réalise une action contient une étoile. Tout d'abord, l'événement `D` n'est pas atteignable, il n'est pas possible d'accéder à celui-ci depuis l'événement primaire `packet`. Également, les événements `F`, `G` et `H` peuvent être supprimés car ils ne mènent pas à un événement qui réalise une action.

2. Une règle feuille est une règle qui ne génère aucun autre événement

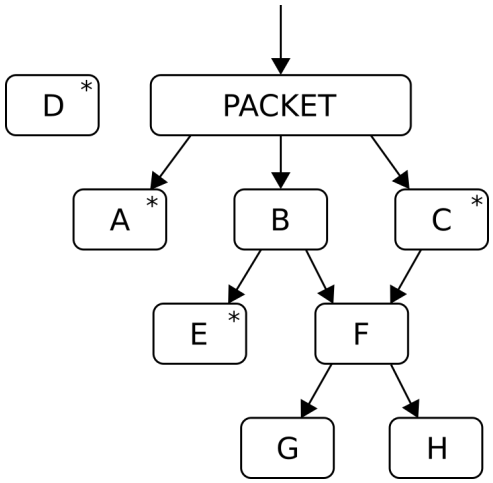


Figure 4.3 – Exemple de graphe d'événements pouvant être optimisé

Langage statiquement typé

Dans la Section 4.1.3, nous avons présenté les différents types utilisés par le langage. La motivation principale de proposer un langage statiquement typé est de pouvoir conduire des analyses sur le fichier de règles et de trouver le plus tôt possible les erreurs réalisées lors du développement de la politique de sécurité. En effet, dès la compilation, nos outils peuvent détecter les incohérences dans le fichier source. De plus, cela évite que du code erroné ne soit déployé sur une sonde et provoque des *crashs* du système.

Terminaison de l'analyse d'un paquet

La dernière propriété de notre langage est d'apporter une preuve de la terminaison du traitement lié à l'analyse d'un paquet réseau. Lors de la phase de compilation, nos outils étudient le graphe des événements et est alors capable de détecter les cycles. Un cycle correspond à une succession de transitions qui fait intervenir au moins deux fois le même nœud. La Figure 4.4 illustre un cycle, qui sera détecté par nos outils. Une fois ces cycles détectés et éliminés, l'analyse d'un paquet se terminera ultimement après avoir effectué toutes les transitions possibles.

Analyse de séquence de paquets

Comme nous l'avons mentionné dans l'état de l'art, certains langages de description ne permettent d'analyser qu'un paquet à la fois. Par exemple, les signatures de Snort décrivent le traitement à réaliser sur un paquet donné et concluent si une intrusion a été détectée pour ce paquet. Cela limite grandement l'analyse, puisque des contournements simples sont possibles, en fragmentant les paquets. D'autre part, pour les protocoles basés sur le concept de connexion, les messages sont échelonnés sur plusieurs paquets, c'est pourquoi il est nécessaire de pouvoir

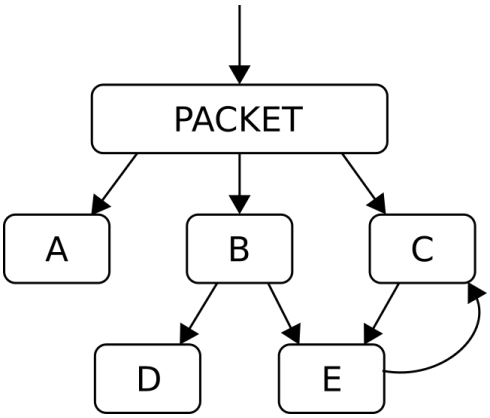


FIGURE 4.4 – Exemple de graphe d'événements impliquant un cycle

conduire une analyse sur plusieurs paquets.

Au travers de DISCUS SCRIPT, il est possible de conduire de telles analyses. En effet, l'utilisation des tables permet de conserver des données contextuelles et de corréler plusieurs paquets. Par exemple, pour le protocole TCP, la table `tcp_connection_table` peut être utilisée pour corréler les informations d'une connexion. Nous présentons un exemple d'une table simplifiée dans la présentation de la syntaxe du langage, dans la prochaine section.

4.2 Éléments de la syntaxe

Dans cette section, nous présentons les éléments de la syntaxe de DISCUS SCRIPT, en détaillant la grammaire et la sémantique associée. Ces présentations sont accompagnées d'exemples. Un fichier de description de règles écrit en DISCUS SCRIPT est composé d'instructions, toutes terminées par le caractère « *;* ». Nous commençons tout d'abord en décrivant les éléments de base du langage, à savoir les instructions et les expressions. Puis nous abordons les concepts clés du langage, à savoir les événements, la recherche de motif et la manipulation des tables.

4.2.1 Avant-propos

Nous présentons la syntaxe de DISCUS SCRIPT de manière traditionnelle, en détaillant la grammaire des différents éléments du langage. Pour que ces descriptions soient lisibles, nous avons distingué les entités lexicales (ou *tokens*) des autres éléments grammaticaux en les écrivant en majuscule. Par exemple, `ENUM` correspond au mot-clé du langage « *enum* », par contre `constant` correspond à la règle grammaticale à laquelle sont associées plusieurs définitions de constantes. Également, certaines entités lexicales simples (éléments de ponctuation, opérateurs, etc.) sont écrites en clair dans les règles, échappées par des guillemets.

4.2.2 Énumération

Les énumérations constituent un type de base du langage, à l'instar des énumérations dans les langages généraux. La grammaire associée à la déclaration de ces énumérations est détaillée dans le Listing 4.3. À la lecture du fichier source, la chaîne de compilation s'assure que les règles sémantiques sont respectées. Notamment, elle s'assure qu'il n'existe pas plusieurs champs portant le même nom et que ces noms ne sont pas déjà déclarés dans le contexte global. Dans le cas où des erreurs de ce type sont rencontrées, des messages sont générés pour alerter l'utilisateur. L'utilisateur a également la possibilité de renseigner les valeurs associées aux champs d'une énumération, à la manière des énumérations en C.

Listing 4.3– Grammaire de la déclaration d'une énumération

```
1 enum_declaration: ENUM STRING '{' enum_field_list '}'
2 enum_field_list: enum_field
3     | enum_field ',' enum_field_list
4 enum_field: STRING
5     | STRING '=' integer_constant
```

Le Listing 4.4 propose plusieurs exemples de déclaration d'énumérations. Dans ces déclarations, l'énumération `tcp_connection_state` permet de donner une représentation simpliste des phases d'existence d'une connexion TCP [3]. Quant à l'énumération `tcp_flags`, elle permet de représenter les différentes valeurs du champ `flags` dans l'en-tête d'un paquet TCP. Les champs déclarés, par exemple `TCP_HS_1` ou `SYN`, sont alors utilisables dans l'ensemble du fichier source.

Listing 4.4– Exemples de déclaration d'énumérations

```
1 enum tcp_connection_state {TCP_HS_1, TCP_HS_2, ESTABLISHED, CLOSED};
2 enum tcp_flags {FIN=1, SYN=2, RST=4, PSH=8, ACK=16, URG=32, ECE=64, CWR=128, NS=256};
```

4.2.3 Constantes globales

Il est possible de déclarer des constantes globales, c'est-à-dire accessibles dans l'ensemble du fichier source. Ces constantes sont sujettes aux mêmes vérifications sémantiques que les énumérations, à savoir la vérification que le nom de la variable est bien unique.

Listing 4.5– Grammaire de la déclaration d'une variable globale

```
1 global_var: LET STRING '=' constant
```

Le Listing 4.5 et 4.6 présentent respectivement la grammaire et des exemples de déclaration d'une variable globale. Une variable globale est en réalité un label associé à une constante, dont

3. L'automate à états TCP étant assez riche, nous nous limitons dans le mémoire à un exemple simplifié

la valeur est connue lors de la compilation. Par exemple, le label `HOME_NET` est associé à la valeur constante de type `ipaddr` qui représente le réseau 192.168.0.0./24. Pour finir, lorsqu'une variable globale est utilisée, le type associé à la constante sera utilisé pour les vérifications sémantiques.

Listing 4.6– Exemples de déclarations de constantes globales

```
let HOME_NET = 192.168.0.0/24;
let SMTP_SERVERS = 192.168.1.0/24;
let FTP_PORTS = [21];
let HTTP_PORTS = [80, 8080];
```

4.2.4 Expressions de base

Le concept principal de notre langage est l'utilisation des événements, qui se déclenchent à l'arrivée d'un paquet réseau ou bien à l'initiative d'un autre événement. Par exemple, il peut être nécessaire d'exprimer des conditions dans ces règles, afin de conduire une analyse plus poussée. L'élément de base de notre langage est l'expression, qui regroupe de nombreuses catégories d'actions. Le Listing 4.7 présente la grammaire des expressions. Nous détaillons ces expressions dans les prochains paragraphes.

Listing 4.7– Grammaire d'une expression

```
expression: atomic_expression
    | match_expression
    | binary_expression
```

Expression atomique

Une expression atomique est une expression qui ne fait intervenir qu'une seule opération ou entité. Le Listing 4.8 illustre la grammaire associée à ces expressions, que nous détaillons ci-dessous.

Listing 4.8– Grammaire d'une expression atomique

```
atomic_expression : variable
    | bitstream_manipulation
    | language_functions
    | constant
    | '(' expression ')'
    | NOT expression
    | MINUS expression
```

Manipulation de bitstream Les bitstreams, qui représentent une séquence de bits dont la taille n'est pas fixée, peuvent être manipulés de manière à extraire des sous parties. Ces manipulations sont illustrées dans le Listing 4.9 et 4.10, qui présentent respectivement la grammaire et des exemples d'utilisations. Dans le cas de ces manipulations, les bornes sont incluses dans l'extraction.

Listing 4.9– Grammaire de la manipulation d'un bitstream

```
1 bitstream_manipulation: variable '[' expression ':' expression ']'
2     | variable '[' ':' expression ']'
3     | variable '[' expression ':' ']'
4     | variable '[' expression ']'
```

Dans les exemples du Listing 4.10, la variable `b` est de type `bitstream`. La première manipulation (ligne 1) consiste à extraire un entier de type `int8`, qui débute à l'indice 32. À la différence de la première manipulation, les deux manipulations suivantes (lignes 2 et 3) extraient une sous partie du bitstream en omettant une borne. Dans ce cas, le début et la fin du bitstream sont utilisés. Par exemple, l'opération `b[:31]` extrait un entier de type `int32` en commençant au début du bitstream `b`. Quant à l'expression `b[128:]`, elle extrait la sous partie du bitstream débutant à l'indice 128 ; comme nous manipulons un bitstream dont la taille n'est pas fixée, le résultat de cette expression sera de type `bitstream`. La manipulation (ligne 4) permet d'extraire le 17ème bit du bitstream `b`. Pour finir, la dernière manipulation (ligne 5) permet d'extraire un entier de type `int64`, en partant de la fin, en démarrant au 128ème bit.

Listing 4.10– Exemples de manipulations d'un bitstream

```
1 ... b[32:40] ...
2 ... b[:31] ...
3 ... b[128:] ...
4 ... b[16] ...
5 ... b[-128:-64] ...
```

Comme la grammaire le suggère, les extractions de bitstreams peuvent être plus complexes, puisqu'il est possible d'utiliser des expressions comme bornes de l'extraction. Cela permet de manipuler des bitstreams en fonction du paquet courant. Ceci est nécessaire pour certains protocoles tels que IPv4 qui possède le champ `options` seulement dans certains cas. Concernant le type du résultat de l'extraction, il est déterminé à la compilation en fonction des opérandes : si les opérandes sont des constantes, notre chaîne de compilation déterminera qu'il s'agit d'un entier. Dans les autres cas, elle considèrera que le résultat est de type `bitstream`.

Concernant les vérifications sémantiques, nos outils s'assurent de la cohérence de l'expression. Notamment, il est vérifié que la variable sujette à l'extraction est bien une expression de type `bitstream` et que les bornes fixées sont bien cohérentes (quand il s'agit de constantes). Dans le cas d'une extraction dynamique, c'est-à-dire qu'au moins une des bornes n'est pas statiquement connue, si l'extraction mène à une incohérence, notre moteur ne poursuit pas l'extraction et

arrête l'analyse de l'événement courant.

Fonctions du langage Bien que nos travaux autour du langage dédié souhaitent s'orienter vers une approche où la majorité des concepts sont exprimables en utilisant le langage, nous avons incorporé des fonctions propres à celui-ci pour les notions complexes mais primordiales. Ces fonctions, qui ont un nombre d'arguments fixé, sont présentées ci-dessous :

- **len(expression)** permet de retourner la taille (en bits) d'une variable de type `bitstream` ;
- **asbig(expression)** et **aslittle(expression)** permettent de convertir un entier, en modifiant son endianness ;
- **net4(expression, expression)** permet de créer une expression de type `ipaddr`, les arguments étant respectivement l'adresse IPv4 et le masque réseau ;
- **net6(expression, expression)** permet, à l'instar de `net4`, de créer une expression de type `ipaddr`, mais pour le protocole IPv6 ;
- **intN(expression)**, où `N` est un entier, permet de transformer un bitstream en un entier de taille `N`.

Les vérifications sémantiques réalisées pour ces fonctions se concentrent sur le type des expressions données en paramètre. Par exemple, nos outils s'assurent que les paramètres de la fonction `net4` sont respectivement de type `int32` et `int5`.

Constantes Les constantes sont des expressions atomiques représentant des expressions dont la valeur est connue à la compilation. Parmi ces expressions, nous pouvons rencontrer :

- `false` et `true` qui représentent chacun un entier de type `int1` ;
- `now` qui désigne une estampille temporelle du moment courant, de type `time` ;
- **les valeurs entières** qui peuvent être exprimées dans les bases décimales ou hexadécimales et dont la longueur est calculée à la compilation ;
- **les valeurs flottantes** ;
- **les représentations d'adresses réseau** (d'une machine ou d'un réseau) de type `ipaddr` ;
- **les listes d'entiers**.

Le Listing 4.6 propose des exemples d'utilisations de listes d'entiers et de représentations d'adresses réseau. Les listes, dont les entiers sont statiquement connus, sont utilisées pour réaliser des opérations ensemblistes, que nous détaillerons dans la section traitant des expressions binaires. Les vérifications sémantiques se concentrent sur les adresses réseau, pour lesquelles nous nous assurons de la cohérence des valeurs.

Autres expressions atomiques D'autres expressions atomiques sont disponibles dans le langage, telles que l'opération unaire ou l'expression parenthésée. Ces expressions représentent les trois dernières lignes de la grammaire illustrée dans le Listing 4.8. Pour les expressions unaires (`not` et `minus`), les vérifications sémantiques se concentrent sur le typage de l'expression sujette à l'opération. Plus précisément, nous nous assurons qu'il s'agisse bien d'un entier.

Pour finir, les variables font référence à des labels déclarés auparavant dans le fichier source. Il peut s'agir de variables globales, de variables locales ou encore de champs d'une énumération. Le type de la variable fait référence au type de l'expression et des vérifications sont réalisées pour s'assurer que la variable existe.

Expression binaire

Les expressions binaires font intervenir deux opérandes séparées par un opérateur. Pour l'ensemble de ces opérations, les vérifications sémantiques se concentrent sur la cohérence des types des opérandes et s'assurent que l'expression résultante est réalisable. La grammaire de ces expressions est présentée dans le Listing 4.11. Chacune de ces expressions est présentée dans les prochains paragraphes.

Listing 4.11– Grammaire des expressions binaires

```
1 binary_expression: arithmetic_expression
2     | comparison_expression
3     | concat_expression
4     | logical_expression
5     | set_expression
```

Expressions arithmétiques, comparatives et logiques Les opérations usuelles, qui manipulent des entiers ou des flottants, sont disponibles dans Discus Script. Permettant d'obtenir de nouvelles valeurs ou d'effectuer des conditions, nous vérifions pour ces expressions que les opérandes sont cohérentes par rapport à l'opération. Notamment, nous vérifions qu'il s'agit bien d'entiers ou de flottants (cela regroupe les types `intN`, `float`, `time` et `enum`).

Concaténation de bitstreams La concaténation de deux bitstreams est possible en utilisant l'opérateur `++`. Cette opération est utile, notamment pour reconstruire des flux tels que ceux du protocole TCP.

Expression ensembliste L'opérateur `in` permet d'effectuer une recherche d'un élément dans un ensemble. Deux cas de figure sont possibles, que nous illustrons dans le Listing 4.12. Le premier usage de cet opérateur (ligne 1) permet de vérifier qu'une valeur est présente dans une liste d'entiers statiquement connus. L'autre usage (ligne 2) permet de vérifier qu'une adresse

appartient à un réseau. Dans les deux cas, nous vérifions que les types des deux opérandes sont cohérents.

Listing 4.12– Exemples d'opérations ensemblistes

```
1 ... port in [80,443] ...
2 ... src_addr in 192.168.0.0/24 ...
```

Recherche de motifs

Les méthodes de détection d'intrusions sont multiples, mais un certain nombre se repose sur la recherche d'un motif dans le paquet en cours d'analyse. C'est notamment une des méthodes principales du moteur de signatures de Snort. Nous proposons également ce concept dans notre langage grâce à l'opérateur `match`. Nous illustrons la grammaire de cette opération dans le Listing 4.13.

Listing 4.13– Grammaire de la recherche de motifs

```
1 match_expression: expression MATCH QUOTED_STRING
2     | expression MATCH QUOTED_STRING AS '(' arg_list ')'
```

Notre opération `match` permet de rechercher un motif dans une expression. Nos vérifications s'assurent que l'expression en question est de type `bitstream` et que le motif recherché est cohérent. Nous offrons la possibilité d'extraire des sous parties si le motif est présent, qui sont exprimées dans le motif par une expression régulière parenthésée. Nous donnons quelques exemples dans le Listing 4.14. L'exemple de la ligne 2 permet d'extraire la version du protocole HTTP, si le motif est trouvé. Le cas échéant, les variables `major` et `minor` sont de type `bitstream` et peuvent être transformées en entier grâce à la fonction du langage `intN()`.

Listing 4.14– Exemples de recherche de motifs

```
1 ... a_bitstream match "GET /" ...
2 ... payload[128:] match "HTTP/(\d).(\d)" as (major, minor) ...
```

Ces recherches de motifs peuvent être utilisées lors des analyses du trafic réseau, qui est géré de manière événementielle. Nous présentons comment écrire ces événements dans la prochaine section.

4.2.5 Gestion des événements

L'élément basique de DISCUS SCRIPT est l'événement, qui réagit à un stimulus précis. L'événement élémentaire, à partir duquel est initiée la détection d'intrusions, est la réception d'un paquet réseau. Un script est alors composé d'un ensemble de règles, qui sont déclenchées suivant les données reçues, pour conduire l'analyse du trafic.

Nous présentons dans le Listing 4.15 la syntaxe utilisée pour déclarer une règle [4]. Nous présentons uniquement dans cette section les éléments de la syntaxe relatifs à la gestion des événements. Les autres éléments, relatifs aux tables, seront présentés par la suite (notamment les instructions des lignes 3 et 5).

Listing 4.15– Grammaire simplifiée de la déclaration d'un événement

```
1 on <event_name>(<arguments>)
2     [ local variables ]
3     [ where <expression> ]
4     [ <for statement> ]
5     [ <event statements> ]
6     [ <ifnone statement> ];
```

Prototype d'un événement

La signature d'un événement, à l'instar des fonctions des langages généraux, est constituée du nom de l'événement et d'un ou plusieurs arguments. Les arguments sont typés, ce qui nous permet de conduire des analyses et de détecter des erreurs de cohérence de typage. Plusieurs règles portant le même nom sont déclenchées par le même stimulus. L'ordre d'évaluation de ces règles n'est pas fixé et dépend de la cible de sécurité, qui peut supporter ou non la parallélisation. Discus Script impose que ces règles partagent la même signature, c'est-à-dire que les règles ont les mêmes paramètres.

Variables locales

Il est possible de déclarer des variables locales dans chacune des règles. Ces variables répondent aux mêmes exigences que les variables globales et les mêmes vérifications sont menées. Notamment, nous vérifions que le nom de la variable est libre et n'interfère pas avec d'autres variables (globales ou non) ou des champs d'énumération. La portée de ces variables est restreinte à l'exécution de la règle. Nous présentons un exemple de déclarations de variables locales dans le Listing 4.16.

Listing 4.16– Déclaration de variables locales

```
1 on packet(bitstream payload)
2     let a = payload[0:8]
3     let b = a + 12
4     (...)
```

4. La grammaire de la déclaration d'un événement est présent dans l'annexe de ce mémoire

Conditions d'exécution

L'exécution d'une règle est conditionnée par la réception d'un événement et d'une clause conditionnelle optionnelle, que l'on peut préciser en utilisant le mot clé `where`. Cette clause permet de réaliser des tests en fonction des paramètres et conditionne l'exécution des clauses suivantes. Nous illustrons l'utilisation d'une clause conditionnelle dans le Listing 4.17.

Listing 4.17– Utilisation d'une clause conditionnelle

```
1 on packet(bitstream payload)
2     where payload[96:111] == 0x800 # Champ EtherType == IPv4
3     (...)
```

Actions réalisables par un événement

Nous avons présenté dans la section précédente comment une règle pouvait être conditionnée par une expression. Si les conditions sont réunies, certaines clauses sont alors exécutées. Nous les présentons dans les prochains paragraphes. Les actions relatives à la manipulation des tables seront présentées dans la Section 4.2.6.

Génération d'événements Notre langage adopte une approche événementielle, c'est à dire qu'un traitement est déclenché par un stimulus précis. L'événement élémentaire est déclenché par l'arrivée d'un paquet réseau et les autres événements découlent de cet événement primaire. Pour cela, nous utilisons la clause `raise`, qui permet de déclencher un autre événement. L'utilisation de cette clause est possible de la manière suivante :

```
raise <event_name>(<parameters>) [in X]
```

Nos outils s'assurent que l'événement déclenché existe et que le type des paramètres correspond à la signature de la règle. Le développeur a la possibilité de retarder le déclenchement de la règle grâce au mot clé `in`, après lequel la durée (en millisecondes) est indiquée. Des exemples sont proposés à la fin de cette section.

Génération d'alertes L'objet d'un système de détection d'intrusions est d'alerter l'administrateur lorsqu'une attaque a été détectée. Notre clause `alert` permet de signaler qu'une action malicieuse est en cours. Le comportement à adopter est découplé de la description des règles et c'est au responsable de la sécurité de décider des actions à réaliser : envoi d'un mail, d'un SMS, exécution d'un script, reconfiguration d'un firewall, etc. La clause `alert` peut être utilisée de la manière suivante :

```
alert(<category>, <message>)
```

La catégorie représente le type d'intrusion détecté. L'usage du caractère « . »permet de raffiner la catégorie, en spécifiant une sous-catégorie. Par exemple, un développeur pourrait créer la catégorie `portscan` et la catégorie `portscan.syn`. Un comportement généraliste serait attribué à la catégorie générale `portscan`, alors qu'un comportement plus spécifique serait exécuté lors des alertes de type `portscan.syn`. Un message accompagne ces alertes, pour spécifier l'intrusion détectée.

Exemples Nous présentons dans le Listing 4.18 plusieurs événements qui utilisent ces actions. L'événement `A` déclenche les événements `B` et `C` grâce à l'action `raise`. Le déclenchement de l'événement `C` est reporté à 500 millisecondes grâce à l'instruction `in 500`. Pour finir, les événements `B` et `C` lèvent une alerte pour indiquer qu'ils ont été déclenchés.

Listing 4.18– Exemple d'événements réalisant des actions

```
1 on A(...)
2     raise B(...)
3     raise C(...) in 500;
4
5 on B(...)
6     alert("Event B", "B has been triggered");
7
8 on C(...)
9     alert("Event C", "C has been triggered")
```

4.2.6 Manipulation de tables

Un des objectifs de notre solution est de faire collaborer les sondes de sécurité. Nous le faisons grâce à la notion de tables, concept emprunté au domaine des systèmes de gestion de bases de données. Les tables, à l'instar des bases de données, contiennent un ensemble d'enregistrements qui peuvent être manipulés dans notre fichier de règles. Nous présentons dans cette section les éléments de syntaxe relatifs à la manipulation de ces tables.

Déclaration d'une table

La déclaration d'un table est similaire à la déclaration d'une structure en C. Une table est caractérisée par un nom et par un ensemble de champs typés. Nos outils s'assurent que les champs portent un nom unique. La grammaire associée à la déclaration de ces tables est présentée dans le Listing 4.19 et un exemple de déclaration de table est illustré dans le Listing 4.2.

Listing 4.19– Grammaire de la déclaration d'une table

```
1 table_declaration: TABLE STRING '{' field_list '}';
2 field_list: field
3     | field field_list ;
4 field: type_specifier field_names ';';
5 field_names: STRING
6     | STRING ',' field_names;
```

Suppression et purge d'enregistrements

Les enregistrements, après avoir été insérés dans les tables, peuvent être supprimés dans deux types de circonstances. La première circonstance est que l'enregistrement a atteint un état obsolète. Nous appelons **suppression** ce type d'opération. La seconde circonstance est plus spécifique aux cibles et se présente quand celles-ci ne possèdent plus assez de mémoire libre. Il est alors nécessaire que le système effectue une opération de **purge** pour libérer de l'espace mémoire.

Ces deux opérations sont représentables au travers de deux instructions dans notre fichier DISCUS SCRIPT. La grammaire de ces instructions est présentée dans le Listing 4.20.

Listing 4.20– Grammaire des instructions de suppression et de purge d'une table

```
1 /* <variable> and <table_name> are STRING tokens */
2 table_deletion: REMOVE <variable> FROM <table_name> WHEN expression
3 table_purge: ON PURGE <table_name> SELECT <variable> WHERE expression
```

Un exemple est illustré dans le Listing 4.21, où nous décrivons une table qui enregistre l'état des connexions TCP. L'instruction de suppression (ligne 7) indique qu'une entrée doit être supprimée dès que son champs `state` est égal à la valeur `CLOSED`. En effet, lorsque cette condition est rencontrée, l'enregistrement devient obsolète car la connexion est fermée. Quant à l'instruction de purge (ligne 8), elle sélectionne les entrées qui ont été créées il y a plus de 60 secondes.

Listing 4.21– Exemple d'instructions de suppression et de purge d'enregistrements d'une table

```
1 table active_connections {
2     ipaddr local_host;
3     ipaddr remote_host;
4     time timestamp; /* Date de creation de la connexion */
5     enum tcp_state state;
6 };
7 remove entry from active_connections when entry.state == CLOSED;
8 on purge action_connections select entry where (entry.timestamp + 60) < now;
```

Insertion de nouveaux enregistrements

Une fois la table déclarée, nous pouvons insérer de nouveaux enregistrements, lorsqu'un événement est déclenché. La clause `insert` est une action qu'un événement peut exécuter et qui permet d'insérer une nouvelle entrée. Elle peut être utilisée de la manière suivante :

```
insert into <table_name> { <assignements> }
```

Nos outils de vérification s'assurent que tous les champs existent et qu'une valeur leur est attribuée. Nous illustrons une insertion d'enregistrements dans le Listing 4.22. Dans cet exemple, nous ajoutons une nouvelle entrée à la table `active_connections` quand nous recevons un paquet TCP dont l'option `flag` indique une nouvelle connexion. Il serait souhaitable d'éviter que cette insertion se produise quand une entrée correspondante existe déjà. C'est ce que nous présentons dans le prochain paragraphe, au travers des parcours de tables.

Listing 4.22– Exemple d'insertion d'enregistrements dans une table

```
1 on tcp_packet(ipaddr src, ipaddr dst, int9 flags)
2     where flags == SYN
3     insert into active_connections {
4         local_host = src;
5         remote_host = dst;
6         timestamp = now;
7         state = TCP_HS_1;
8     };
```

Parcours d'une table

La recherche d'enregistrements d'une table est réalisable dans l'exécution d'une règle de sécurité. Cela permet d'accéder à des informations contextuelles et permet de compléter l'analyse. Ce parcours est associé aux instructions `for` et `ifnone` du Listing 4.15.

La première partie de cette recherche consiste à parcourir une table. Nous réalisons ce parcours grâce à la clause `for`, qui peut être utilisée de la manière suivante :

```
for [all|first] <variable> in <table_name> with <condition>
```

Cette clause permet de rechercher une entrée d'une table qui répond aux conditions fixées. Le parcours d'une table s'arrête :

— à la première occurrence d'un enregistrement qui répond aux conditions si le mot clé `first` est utilisé ;
— à la fin de la table si le mot clé `all` est utilisé.

Le Listing 4.23 illustre le parcours d'une table, à la recherche d'un enregistrement. Cette règle recherche la première entrée de la table `active_connection` qui corresponde aux paramètres de

l'événement `tcp_packet`. Dans cet exemple, nous nous arrêtons à la première occurrence, car nous considérons qu'une seule entrée existe pour un couple source/destination donné.

Listing 4.23– Exemple de parcours de table

```
1  on tcp_packet(ipaddr src, ipaddr dst, int9 flags)
2      for first entry in active_connections with entry.local_host == src
3                      and entry.remote_host == host
4          /* Update entry */
5          update entry.timestamp = now;
```

Il est probable qu'un parcours de table ne parvienne pas à trouver d'enregistrement correspondant à la condition. L'utilisation de l'instruction `ifnone` permet d'exécuter des instructions dans ce cas de figure. Le Listing 4.24 illustre un exemple où nous cherchons à compter le nombre de paquets rencontrés par protocole TCP au travers de la table `tcp_stats`. Lorsqu'une nouvelle connexion TCP s'ouvre, nous regardons si nous trouvons une entrée correspondante dans la table et le cas échéant, nous mettons à jour le compteur. Dans le cas contraire, l'instruction `ifnone` (ligne 10) spécifie qu'il faut insérer une nouvelle entrée et lever une alerte pour signaler qu'un nouvel enregistrement a été ajouté.

Listing 4.24– Exemple de parcours de table

```
1  table tcp_stats {
2      int16 port;
3      int32 counter;
4  };
5
6  on tcp_packet(..., int16 dport, ..., int9 flags, ...)
7      where flags == SYN
8      for first entry in tcp_stats with entry.port == dport
9          update entry.counter += 1
10     ifnone
11         insert into tcp_stats {
12             port = dport;
13             counter = 1;
14         }
15         alert("TCP Stats", "New entry");
```

Cette instruction se révèle très pratique lorsque l'on souhaite écrire une règle qui fait à la fois l'insertion de données (lorsqu'elles n'existent pas encore) ou leur mise à jour. Les deux prochaines sections proposent des exemples du langage, où des parcours de tables sont réalisés de cette manière.

4.3 Cas d'étude : détection d'injection de scripts

Dans cette section, nous cherchons à détecter une injection de scripts, qui consiste à envoyer du texte qui sera interprété et exécuté par la machine ciblée. Dans un premier temps, nous présentons ce type d'attaque et les méthodes de détection, puis nous détaillons l'implémentation en Discus Script d'une technique de détection.

4.3.1 Description de l'attaque et d'une contre-mesure basique

Une injection de scripts est une attaque courante sur les protocoles web, où une personne malicieuse cherche à faire exécuter un morceau de code par le serveur ciblé. Cette attaque consiste à envoyer le code au serveur et à parvenir à lui faire exécuter. Le code en question peut être envoyé en clair ou manipulé de sorte à tromper les outils de détection. Si l'attaquant parvient à faire exécuter son code, les conséquences peuvent être désastreuses : escalade de privilèges, propagation de vers, fuite d'informations secrètes, etc.

Considérons pour cet exemple qu'un site web est ciblé par un attaquant qui désire se connecter sur ce dernier avec le compte de l'administrateur. Pour cela, il va utiliser le formulaire de connexion, remplir les champs *utilisateur* et *mot de passe* et envoyer sa requête au serveur. La plupart des sites internet utilisent une base de données pour stocker les informations relatives aux utilisateurs, on peut donc supposer qu'une requête SQL est réalisée suite au traitement des données reçues. Le Listing 4.25 présente un exemple de requête, qui vérifie que le couple utilisateur/mot de passe existe.

Listing 4.25– Requête SQL vérifiant l'existence d'un utilisateur

```
1 SELECT *
2 FROM users
3 WHERE username = 'Username'
4 AND password = 'Password'
```

Dans ce cas de figure, l'injection de code est possible en envoyant des données malicieuses dans les champs du formulaire de connexion. Ces données seront transmises au serveur et il les utilisera pour vérifier qu'un utilisateur existe. Le Listing 4.26 présente une injection possible, pour laquelle l'utilisateur malveillant a utilisé la valeur `Password' OR '1'='1` comme mot de passe. La condition `'1'='1'` étant vraie dans tous les cas, quelque soit le résultat de la comparaison du mot de passe, la requête considèrera qu'il s'agit bien de l'utilisateur `Username` en question.

Listing 4.26– Injection SQL

```
1 SELECT *
2 FROM users
3 WHERE username = 'Username'
4 AND password = 'Password' OR '1'='1'
```

La détection de ces injections ne peut être décrite de manière automatique, car il est nécessaire d'écrire une signature pour chacun des motifs malveillants. Si une telle signature existe, pour qu'une solution de sécurité puisse détecter cette attaque, elle doit chercher le motif malveillant dans les paquets à analyser.

4.3.2 Description de l'attaque en DISCUS SCRIPT

La détection de cette injection SQL peut être réalisée en analysant le contenu des champs du protocole HTTP de la requête du client, lorsque ce dernier envoie le contenu du formulaire. Le listing 4.27 présente la règle basée sur l'événement `http_request`, déclenché lorsqu'un client envoie une requête HTTP à un serveur web. L'instruction `match` nous permet de vérifier si le motif est présent dans la requête HTTP du client, le cas échéant, une alerte est levée pour signaler l'attaque à l'administrateur.

Listing 4.27– Détection d'une injection SQL

```
1 on http_request(ipaddr client, ipaddr server, ..., bitstream payload)
2     where payload match "'\s+OR\s+'1'='1"
3     alert("sqli", "Injection SQL detected");
```

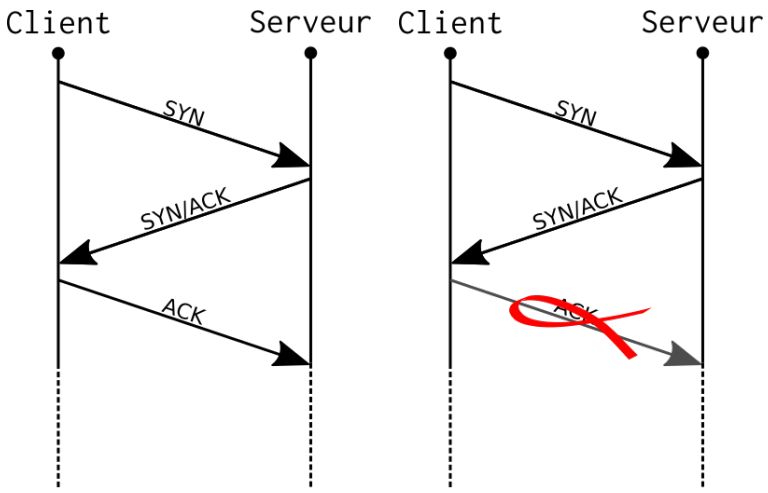
4.4 Cas d'étude : détection de SYN flood

Dans cette section, nous présentons un cas d'utilisation de notre langage DISCUS SCRIPT, dans lequel nous écrivons plusieurs règles pour détecter l'attaque appelée SYN Flood. Dans un premier temps, nous présentons l'attaque et les manières de la détecter. Ensuite, nous détaillerons les règles pour pouvoir détecter cette attaque.

4.4.1 Présentation de l'attaque SYN Flood

L'attaque SYN flood, traduit littéralement par *inondation de SYN*, est une attaque réseau qui vise à détériorer les performances de la cible, en provoquant un déni de service. Le vecteur de propagation de cette attaque est le protocole TCP, plus particulièrement la phase d'initialisation de la connexion TCP entre le client et le serveur.

Un échange normal d'initialisation, également appelé *three-way handshake*, est constitué de trois étapes dans le cas où la connexion peut s'établir. Nous présentons ces trois étapes dans la Figure 4.5a. Dans cette figure, le client démarre tout d'abord la tentative de connexion par l'envoi d'un paquet TCP avec le *flag* `SYN`. Si la connexion est possible avec le serveur, ce dernier répond via un paquet avec les *flags* `SYN` et `ACK`. Pour finir, le client confirme l'ouverture de la connexion par un ultime paquet, contenant le flag `ACK`.



(a) Connexion TCP réussie (b) Attaque SYN Flood

FIGURE 4.5 – Initialisation d'une connexion TCP

Dans le cas d'une attaque SYN flood, l'attaquant réalise une tentative de connexion et omet, de manière volontaire, d'envoyer le dernier paquet. Cette attaque est représentée dans la Figure 4.5b. Du point de vue du serveur, il n'est pas possible de déterminer si le dernier paquet a été retardé, ou est manquant, c'est pourquoi le serveur doit patienter. Durant cette période, le serveur conserve un certain nombre d'informations en mémoire relatif à la connexion en cours d'ouverture. Individuellement, une occurrence de cette attaque est inoffensive, l'espace mémoire occupé étant assez faible. Cependant, une attaque SYN flood est constituée d'un grand nombre d'occurrences de cette attaque, ce qui provoque finalement une consommation mémoire considérable. La conséquence directe de cette consommation mémoire est de provoquer un déni de service, voire même d'arrêter de fonctionner. Lorsque cette attaque est réalisée par plus d'un attaquant, on parle de déni de service distribué, ce qui complique la détection de l'attaque, détaillée dans le prochain paragraphe.

La détection de l'attaque ne peut pas être décrite au travers d'une règle basique de sécurité. En effet, comme nous l'avons présentée précédemment, une occurrence de l'attaque est légitime aux yeux du protocole TCP. C'est pourquoi il n'est pas possible de détecter cette attaque en ne considérant qu'un seul paquet. De nombreuses techniques existent [99, 83, 50] pour détecter le SYN flood. Dans ce mémoire, nous présentons une méthode de détection similaire à celle que proposent Wang et al. [99], qui consiste à dénombrer le nombre de connexions. Si ce nombre excède un seuil fixé, on peut alors en conclure qu'une attaque est en cours. Nous implémentons cette méthode de détection dans la prochaine section.

4.4.2 Implémentation en DISCUS SCRIPT

L'implémentation des règles pour détecter cette attaque fait intervenir plusieurs éléments du langage : les tables, pour stocker les informations relatives aux tentatives de connexion, et les événements, pour insérer les informations dans ces tables. Nous présentons l'utilisation de ces concepts dans les paragraphes suivants.

Table d'informations

Les informations nécessaires à la détection de cette attaque, telles que l'adresse IP de l'attaquant et de la cible ou la date du dernier événement suspect, seront stockées dans la table `syn_flood_t`. La définition de la table est présentée dans le Listing 4.28. Pour illustrer un cas d'utilisation simple du langage, nous nous cantonnerons ici à associer l'adresse source et l'adresse destination de la tentative d'attaque. Ceci nous permettra de détecter une attaque provenant d'un attaquant. Cependant, nous ne pourrons pas détecter les attaques distribuées, puisque plusieurs adresses sources d'attaques sont utilisées. Le champ `state` nous permet d'identifier dans quelle phase de détection nous sommes : détection en cours ou détection terminée.

Listing 4.28– Déclaration de la table syn_flood_t

```
1 enum syn_flood_t {SF_IN_PROGRESS, SF_FINISHED};
2
3 table syn_flood_t {
4     ipaddr client, server;
5     int32 counter;
6     enum syn_flood_state state;
7 };
```

Nous illustrons dans le Listing 4.29 les instructions relatives à la gestion des tables, notamment les conditions de suppression et de sélection d'enregistrements lors d'une purge. Pour cette table, nous supprimons une entrée lorsque la solution a détecté une intrusion, autrement dit quand le champ `state` indique que la détection est terminée. En ce qui concerne la purge, nous choisissons de supprimer les entrées dont le compteur est inférieur à un seuil fixé.

Listing 4.29– Gestion des enregistrements de la table syn_flood_t

```
1 remove entry from syn_flood_t when entry.state == SF_FINISHED;
2 on purge syn_flood_t select entry where entry.counter < 10;
```

Événement TCP

La table que nous avons précédemment présentée sera manipulée principalement par l'événement `tcp_packet`, qui est déclenché lorsqu'un paquet TCP est analysé par la solution. Plusieurs

manipulations seront nécessaires, notamment :

— l'**insertion** d'une nouvelle entrée lorsque nous rencontrons un nouveau couple client/serveur TCP ;

— la **mise à jour** d'une entrée lorsque celle-ci existe ;

— la **détection** qu'une attaque est en cours, une fois que le compteur dépasse un seuil fixé.

Lors d'une nouvelle connexion TCP, l'insertion et la mise à jour des entrées est réalisée par la même règle Discus Script, illustrée dans le Listing 4.30. Dans cette règle, nous vérifions tout d'abord (ligne 2) qu'il s'agit effectivement d'une nouvelle connexion sur le point d'être initialisée. Ensuite, nous cherchons un enregistrement qui correspond à la connexion courante (ligne 3). Dans le cas où un enregistrement existe, nous le mettons à jour et appelons un autre événement qui se chargera de la détection de l'attaque. Si le couple client/serveur est inconnu, nous créons un nouvel enregistrement grâce à l'instruction `insert` (ligne 14).

Listing 4.30– Manipulation de la table lors d'une nouvelle connexion TCP

```
1  on tcp_packet(ipaddr src, ipaddr dst, ..., int9 flags, ...)
2      where flags == SYN | ACK
3      /* Look for a corresponding entry in syn_flood_t */
4      for first entry in syn_flood_t with entry.server == src
5                          and entry.client == dst
6                          and entry.state == SF_IN_PROGRESS
7          /* Updating table */
8          update entry.counter += 1
9          /* Raise an event that will check if the threshold is reached */
10         raise syn_flood_check(dst, src)
11     ifnone
12         /* Adding a new entry */
13         insert into syn_flood_t {
14             server = src;
15             client = dst;
16             counter = 1;
17             state = SF_IN_PROGRESS;
18         };
```

La règle précédente se charge de créer des enregistrements et de les mettre à jour lorsqu'une nouvelle connexion TCP est détectée. Nous illustrons dans le Listing 4.31 la règle qui met à jour un enregistrement lorsqu'une connexion est fermée [5].

5. Pour la simplicité de l'exemple, nous considérons qu'une connexion TCP se termine par un paquet contenant le flag FIN provenant du serveur

Listing 4.31– Manipulation de la table lors de la fermeture d'une connexion TCP

```
on tcp_packet(ipaddr src, ipaddr dst, ..., int9 flags, ...)
    where flags == FIN
    for first entry in syn_flood_t with entry.server == src
                and entry.client == dst
                and entry.counter > 0
        /* Update entry */
        update t.counter -= 1;
```

Pour finir, l'événement `syn_flood_check` se charge de la détection d'une attaque, en comparant le compteur à un seuil fixé. La règle présentée dans le Listing 4.32 considère qu'une attaque est en cours si plus de 50 connexions existent simultanément.

Listing 4.32– Détection de l'attaque SYN Flood

```
on syn_flood_check(ipaddr client, ipaddr server)
    for first entry in syn_flood_t with entry.client == client
                and entry.server == server
                and entry.counter >= 50
        /* Generating an alert */
        alert("dos", "SYN Flood detected")
        /* Updating entry */
        update entry.state == SF_FINISHED;
```

4.5 Synthèse

Dans ce chapitre, nous avons présenté DISCUS SCRIPT, un langage dédié dont l'objectif est de décrire des règles de sécurité réseau. Basé sur une programmation événementielle, le langage s'appuie sur plusieurs concepts pour conduire ses analyses sémantiques, tels que le typage fort (les variables sont statiquement typés) ou l'analyse des graphes d'événements. L'utilisation des tables (structures de données) permet de conserver des informations contextuelles utilisées lors des analyses du trafic. Pour finir, nous avons montré dans les Sections 4.3 et 4.4 que notre langage est capable d'exprimer des règles dont l'analyse se concentre sur un paquet réseau (détection d'injection de scripts) ou sur une séquence de paquets (détection du SYN flood).

Après l'élaboration d'un langage dédié, il est important d'évaluer son efficacité. C'est ce que nous proposons dans le prochain chapitre, où nous étudions l'expressivité de DISCUS SCRIPT, sa robustesse et ses performances. Ces études proposent également des comparaisons avec des solutions de l'état de l'art, afin de se positionner par rapport à l'existant.

5

Évaluation de Discus Script

Notre contribution majeure est la proposition d'un langage dédié, Discus Script, qui permet de décrire des règles de sécurité de manière événementielle. La collaboration est possible en utilisant la notion de tables, assimilables aux tables des bases de données. Dans ce chapitre, nous menons des études pour évaluer notre langage, en utilisant des méthodes de la littérature. Nous distinguons trois méthodes dans la Section 2.3 de l'état de l'art :

Expressivité L'étude est concentrée sur la capacité du langage à exprimer les concepts du domaine ciblé. Cela consiste notamment à se comparer aux autres solutions et à définir les notions que le langage est capable d'exprimer ou non.

Robustesse Cette étude se focalise sur la robustesse du langage et sa suite d'outils. Un tel langage est dédié à un domaine particulier, il est par conséquent capable de mener des analyses fines et d'assurer la cohérence du code source écrit. Cette étude peut consister à confronter le langage à des situations problématiques et à analyser son comportement.

Performances La dernière étude que nous avons distinguée s'intéresse aux performances du langage. Puisque le langage est spécifique à un domaine, il doit être capable d'offrir une vision optimisée de l'ensemble des règles. Nous pouvons étudier ici les performances de la solution issue de la compilation du fichier de règles.

5.1 Étude de l'expressivité de Discus Script

Les raisons de la création d'un langage dédié sont nombreuses, mais la principale provient du fait que les langages généraux ne sont pas capables de décrire les problèmes d'un domaine de manière concise et facile, du point de vue du développeur. D'autre part, dans notre cas, il n'est pas envisageable d'espérer d'un développeur qu'il possède une expertise dans les différents domaines ciblés, à savoir la sécurité réseau et le développement de systèmes embarqués ou de circuits électroniques.

Dans cette section, nous nous intéressons à l'expressivité de Discus Script, une caractéristique qui permet d'évaluer un langage dédié. En effet, l'objectif d'un langage dédié est de fournir un moyen simple d'exprimer des concepts propres au domaine ciblé. Dans un premier temps, nous montrons que notre langage est capable d'exprimer autant de concepts que le langage proposé par Snort, reconnu par la communauté scientifique. Puis, nous présentons les notions qui ne sont pas adressées par les langages existants de la littérature, mais que nous pouvons exprimer en utilisant Discus Script.

5.1.1 Traduction de règles vers Discus

La première étape de notre étude consiste en une vérification que les concepts exprimables dans les langages existants le sont aussi dans notre langage. Cela nous permet de valider que notre solution est aussi expressive que les langages de la littérature. Pour effectuer cela, nous

nous sommes intéressés au langage proposé par Snort, largement utilisé par la communauté et également populaire dans la littérature. Nous avons créé un outil qui traduit les règles Snort vers notre langage. Pour tester notre outil, nous avons utilisé un fichier provenant de la communauté Snort, composé de plus de 600 signatures. Nous présentons cette traduction dans les paragraphes suivants.

Construction d'une signature Snort

L'analyse d'un paquet réalisée par Snort fait intervenir un certain nombre de modules (écrits en C) ainsi qu'un moteur de signatures. Ce moteur applique séquentiellement les signatures au paquet réseau courant. La majorité des règles disponibles proviennent de la grande communauté Snort. Une signature Snort est décomposée de la manière suivante :

```
<action> <protocol> <src> <sport> <direction> <dst> <dport> ( <options> );
```

L'en-tête d'une règle Snort correspond à tous ces éléments, exceptée la partie `options`. Le champ `action` permet d'indiquer l'action à réaliser si les conditions sont réunies. Différentes actions sont possibles, telles que `alert` et `log` qui vont respectivement lever une alerte ou ajouter une ligne dans le fichier de logs. D'autres actions, telles que `pass`, `drop` ou encore `reject` permettent de court-circuiter le transit du paquet. Ces dernières sont très peu utilisées dans les fichiers de la communauté et sont éloignées de la philosophie des IDS, qui se concentrent sur la détection et non la prévention. Le champ `protocol` permet d'indiquer pour quel protocole la règle va s'appliquer. Pour le moment, les protocoles TCP, UDP, ICMP ou IP sont supportés par Snort. Les développeurs de Snort prévoient que d'autres protocoles seront disponibles prochainement, tels que ARP, OSPF ou encore RIP. Les champs `src`, `dst`, `sport` et `dport` permettent d'indiquer les hôtes ou réseaux sur lesquels la signature va s'appliquer. Quant au champ `direction`, il permet de discriminer la direction du paquet, que ce soit de la source vers la destination (`->`), de la destination vers la source (`<-`), ou bien les deux (`<>`).

Le corps d'une règle est constitué du champ `options`. Cette partie correspond aux conditions à appliquer sur le paquet. Si les conditions sont réunies, alors la règle déclenchera l'action inscrite dans le champ `action`. Nous détaillons davantage les options dans les prochains paragraphes.

Le Listing 5.1 propose un exemple d'en-tête de règle Snort. Dans cet exemple, la règle s'applique pour les paquets TCP, qui sont dirigés vers le réseau `192.168.0.0/24` pour les ports 80 et 443. La règle génère une alerte lorsque le motif « *GET* »est trouvé dans le paquet. Nous discutons de la traduction de ces règles dans la prochaine section.

Listing 5.1– Exemple de règle Snort

```
1 alert tcp any any -> 192.168.0.0/24 [80,443] (content:"GET");
```

Traduction de l'en-tête d'une règle Snort

La traduction de l'en-tête d'une signature se repose sur les expressions et événements de base. Puisqu'une règle Snort ne s'applique qu'à un seul paquet, une règle Snort est traduite en un événement Discus Script. Le fichier Snort étudié, de plus de 600 règles, est intégralement composé de règles ayant pour action de générer une alerte. La clause `alert` de Discus Script sera utilisée pour traduire l'action en question. En ce qui concerne le protocole, notre langage est fourni avec une librairie de base contenant les protocoles élémentaires. Par exemple, pour une règle Snort portant sur le protocole TCP, sa traduction en Discus Script se basera sur l'événement `tcp_packet`. La traduction de la partie réseau (adresses et ports) est réalisée dans la clause `where`, où des opérations d'appartenance à un réseau sont générées.

Le Listing 5.2 illustre la traduction de l'en-tête de la règle présentée dans le Listing 5.1. Seul l'en-tête est traduit dans cet exemple. La prochaine section se concentre sur la traduction des options des signatures Snort.

Listing 5.2– Traduction de l'en-tête d'une règle Snort

```
1 on tcp_packet(ipaddr src, int16 sport, ipaddr dst, int6 dport, ...)
2     where dst in 192.168.0.0/24 and
3           dport in [80,443]
4     alert("snort.trad", "A message");
```

Traduction du corps d'une règle Snort

Le corps d'une signature Snort est constitué d'une ou plusieurs options, qui s'écrivent de la manière suivante : `<name>[:<values>];`. Une option peut être spécialisée en ajoutant des *modifiers*, qui précèdent l'utilisation d'une option. Par exemple, la séquence d'options :

```
http_header;content:"GET"
```

permet de rechercher le motif uniquement dans l'en-tête HTTP d'un paquet. Puisque notre outil de traduction a pour objectif d'être une preuve de concept, nous nous sommes focalisés sur la traduction des options présentes dans le fichier source Snort. Sans dénombrer les différents *modifiers*, le fichier de règles est composé d'une quinzaine d'options principales, pour lesquelles nous décrivons le processus de traductions dans les prochains paragraphes.

Méta-données Plusieurs options correspondent à des informations relatives à la détection de l'attaque. Nous utilisons ces méta-données pour enrichir la traduction de la signature et que sa lecture soit plus aisée. Par exemple, nous utilisons les champs `sid` (identifiant de signature), `msg` ou `reference` lorsque des alertes sont générées. De cette manière, l'alerte fera référence à la signature Snort initiale et le message explicitera l'intrusion détectée.

Recherche de motifs L'opération de recherche de motifs est réalisable via deux options dans les signatures Snort : `content` et `pcre`. L'option `content` recherche un contenu alors que l'option `pcre` recherche un motif exprimé par une expression régulière. Il faut noter que ces opérations de recherche, dans le cas où le motif est trouvé, décalent les prochaines recherches, en positionnant un pointeur après l'apparition du motif. Pour illustrer cette notion de décalage, le Listing 5.3 propose un exemple de règle Snort qui implique plusieurs recherches de motifs. Les deux règles (lignes 2 et 3) sont appliquées sur les données de la ligne 5. La première règle réunit les conditions, puisque les deux contenus sont trouvés dans les données, dans l'ordre spécifié. Par contre, la seconde règle ne réunit pas les conditions, bien que l'ensemble des motifs soit présent dans les données. En effet, la deuxième recherche de motifs a décalé le pointeur de recherche, qui se trouve maintenant à `.com`. La dernière recherche de motifs ne peut donc pas réussir.

Listing 5.3– Recherche de motifs dans une règle Snort

```
1 /* Snort rules */
2 alert tcp any any -> any any (content:"GET"; content:"HTTP";);
3 alert tcp any any -> any any (content:"GET"; content:"example"; content:"www";);
4 /* Data */
5 GET /index.html HTTP/1.1\r\nHost: www.example.com
```

Pour traduire ces recherches de motifs, nous utilisons la clause `match`. Nous gérons le décalage en capturant la partie succédant au motif, qui sera utilisée pour la prochaine opération de recherche. La traduction de la première règle Snort (ligne 2 du Listing 5.3) est présentée dans le Listing 5.4. Les variables `tail_0` et `tail_1` sont extraites des clauses `match` et permettent de décaler la recherche de motifs.

Listing 5.4– Traduction des recherches de motif

```
1 on tcp_packet(..., bitstream payload, ...)
2     where payload match "GET(.*)" as (tail_0)
3           and tail_0 match "HTTP(.*)" as (tail_1)
4     alert("...", "...");
```

Détection par seuil En plus de l'analyse d'un paquet réseau, Snort offre la possibilité de détecter une intrusion lorsqu'un événement suspect se produit un certain nombre de fois dans une période fixée. L'alerte est alors générée une fois que le nombre d'événements suspects dépasse un certain seuil. L'option permettant cette méthode de détection est `detection_filter`, qui s'utilise de la manière suivante :

```
detection_filter: track <by_src|by_dst>, count <c>, seconds <s>;
```

Dans le Listing 5.5, nous présentons une règle Snort qui utilise cette option. Dans ce cas, l'alerte sera levée lorsque 30 paquets réseau contenant le motif « *GET* »transiteront par la solution de sécurité, en moins de 10 secondes.

Listing 5.5– Signature Snort utilisant la détection par seuil

```
1 alert tcp any any -> any any (content:"GET"; detection_filter: track by_src, count 30, seconds 10);
```

Jusqu'à présent, notre outil traduit une règle Snort en un événement Discus Script. Pour la traduction de cette option, nous avons besoin d'éléments supplémentaires. Dans un premier temps, nous avons besoin de déclarer une table, qui stockera les informations relatives aux événements suspects. Ensuite, il nous faut un ensemble d'événements qui manipule cette table, notamment pour mettre à jour les enregistrements ou déclencher l'alerte quand le seuil est dépassé. Nous présentons dans le Listing 5.6 la table que notre outil aurait généré pour la signature Snort précédemment illustrée. Dans cette traduction, la table `threshold_filter` est constituée des champs `host`, `count` et `last`, qui représentent respectivement la machine suspectée, le nombre d'événements et une estampille du dernier événement. Le champs `state` permet quant à lui de déterminer si l'enregistrement correspond à une analyse en cours ou non. Notamment, ce champ est utilisé pour déterminer si une entrée de la table peut être supprimée. Pour finir, la purge de la table s'effectue sur les entrées qui datent de plus de 60 secondes.

Listing 5.6– Table et opérations générées pour la détection par seuil

```
1 enum threshold_state {IN_PROGRESS,DETECTED};
2 table threshold_filter {
3     ipaddr host;
4     int16 count;
5     time last;
6     enum threshold_state state;
7 };
8 remove entry from threshold_filter when entry.state == DETECTED;
9 on purge threshold_filter select entry where (now - entry.last) > 60;
```

La table sera alors manipulée au travers d'événements, déclenchés à la réception des paquets réseau. Nous présentons ces événements dans le Listing 5.7. Le premier événement (ligne 1) vérifie que le motif est présent dans les données. De plus, il parcourt les enregistrements de la table et cherche une entrée correspondante à la machine `src`. Dans le cas où une entrée existe, les champs sont mis à jour. Autrement, une nouvelle entrée est ajoutée à la table. Le second événement (ligne 15) est appelé lorsqu'une entrée est mise à jour. Elle vérifie si le seuil est dépassé et lève une alerte le cas échéant.

Listing 5.7– Événements générés pour la détection par seuil

```
on tcp_packet(ipaddr src, ..., bitstream payload, ...)
    where payload match "GET(.*)" as (tail_0)
    for first entry in threshold_filter with entry.host == src
        update entry.count = entry.count + 1
        update entry.last = now
        alert threshold_check(src)
    ifnone
        insert into threshold_field {
            host = src;
            count = 1;
            last = now;
            state = IN_PROGRESS;
        };

on threshold_check(ipaddr host)
    for first entry in threshold_filter with entry.host == host and entry.count >= 30
        update entry.state = DETECTED
        alert("...", "...")
```

Suivi de connexion L'option `flow` permet d'appliquer des signatures Snort uniquement si le paquet est dans un certain état de connexion. Pour traduire ces règles, nous nous sommes reposés sur la table `tcp_connections` qui recense toutes les connexions ouvertes. Cette table, disponible dans la librairie de base de Discus Script, permet d'établir si une connexion est ouverte ou non, et permet également d'identifier quel hôte est client ou serveur. La traduction consiste donc à vérifier qu'une entrée existe (ou non) dans la table.

Options relatives aux modules Snort Certaines options ne sont pas traduites par notre outil, parce que leur fonctionnement se concentre sur la communication interne avec les modules Snort, développés en C. Par exemple, l'option `flowbits` communique avec le module `Session`, qui se charge du suivi des sessions. Seule une minorité des règles du fichier étudié (à savoir, un peu plus de 600 signatures Snort) utilise ces options. Dans la prochaine section, nous discutons de la traduction de ce fichier dans sa globalité.

Synthèse

Cette phase de traduction de règles existantes était pour nous une étape importante, puisqu'elle nous a permis d'établir que notre langage est au moins aussi expressif qu'un langage admis par la communauté. Nous avons éprouvé notre outil en le faisant traduire un fichier issu de la communauté, composé de 627 règles de sécurité. Parmi celles-ci, seule 19 règles n'ont pu être

traduites, car ces règles utilisaient des options intimement liées aux modules Snort développés en C. Au final, notre outil a transformé ces 608 règles Snort en 628 règles Discus Script, dont la majorité repose sur une translation entre ces deux langages.

5.1.2 Expressivité du langage

La section précédente se concentrait sur la traduction de règles d'un langage de l'état de l'art, reconnu comme étant une solution idéale pour détecter les intrusions réseau. Nous avons montré, au travers de cette traduction, que notre langage est aussi expressif que Snort. Dans cette section, nous montrons que notre langage permet d'exprimer des notions supplémentaires, contrairement à d'autres langages existants de la littérature. Pour cela, nous présentons plusieurs notions relative à la sécurité réseau et discutons de leur support ou non pour plusieurs langages existants de l'état de l'art. Nous nous limiterons dans les prochains paragraphes à comparer Discus Script, Snort, NeMode et Bro. Nous proposons toutefois dans la synthèse un tableau comparatif avec l'ensemble des solutions présentées dans l'état de l'art.

Méthodes de détection

Les langages permettent de spécifier des règles qui décrivent les étapes à suivre pour détecter une intrusion. Comme nous l'avons présenté dans l'état de l'art, il existe deux techniques majeures pour détecter une intrusion. La première, la détection d'anomalies, consiste à décrire le comportement normal d'un système. Tout comportement déviant de la normalité sera alors considéré comme malicieux. La deuxième, la détection par signatures, se repose sur la description des comportements malicieux, présentant les différents éléments à rechercher pour convenir qu'une intrusion est en cours.

Chacune des deux techniques possède des avantages et des inconvénients. Notamment, la détection d'anomalies peut générer beaucoup de faux-positifs, puisqu'il est difficile d'établir précisément le comportement normal d'un système, dans la théorie et les implémentations que les constructeurs choisiront d'appliquer. Quant à la détection par signatures, il est nécessaire d'avoir une base de données de règles constamment à jour, car une intrusion ne peut être détectée que si la description associée est connue du système de détection.

En ce qui concerne les langages de descriptions de règles de sécurité, tous les langages étudiés se concentrent sur la détection par signatures, y compris Discus Script. La description de règles de sécurité ne se prête pas forcément à la détection d'anomalies, qui se concentre sur la modélisation d'un comportement normal, dont la conception est réalisée une fois pour toute. Généralement, la détection d'anomalies est exprimée directement dans l'implémentation de la solution de sécurité, à l'instar des modules de Snort, écrits en C.

Mécanisme de détection à états

La détection d'intrusions peut être réalisée à plusieurs niveaux de granularité. La plus faible granularité se focalise sur l'élément primaire à analyser, le paquet réseau. Dans le cas d'un paquet, cela consiste à analyser, ses particularités et les données qu'il transporte. Cette analyse élémentaire est supportée par l'ensemble des solutions étudiées dans cette section. La granularité supérieure permet d'analyser non plus un paquet réseau, mais une séquence de paquets. Cette technique de détection permet d'analyser plus précisément les protocoles à états, tels que TCP et UDP (protocoles binaires) ou encore HTTP et FTP (protocoles textes). Pour finir, le dernier niveau d'analyse concerne l'aspect collaboratif, que nous présentons dans la prochaine sous-section. Nous discutons dans la suite de ces paragraphes de la capacité des langages à exprimer des règles analysant des protocoles à états, c'est-à-dire conduire une analyse sur une séquence de paquets connexes.

Notre solution, Discus Script, permet de décrire des règles analysant à la fois les paquets réseaux (de manière individuelle) ainsi que des séquences de paquets. L'événement élémentaire de notre langage est déclenché à la réception d'un paquet réseau. Les différentes actions réalisables lors de l'exécution d'un événement permettent notamment de manipuler les tables, les structures de données que l'on utilise alors pour relier plusieurs événements. Par exemple, la table `active_connections` décrite dans le Listing 4.21 illustre une table utilisée par plusieurs événements pour recouper les informations relatives à différents paquets réseaux. De cette manière, il est possible de décrire un mécanisme de détection d'intrusions opérant sur plusieurs paquets réseaux.

Quant au langage Snort, il ne supporte pas la description de règles opérant sur plusieurs paquets. En effet, une règle Snort est associée à la détection d'une intrusion pour un paquet. La solution Snort, quant à elle, peut détecter des intrusions se déroulant sur plusieurs paquets et états, notamment grâce à l'utilisation des modules écrits en C. Ces mécanismes ne pouvant pas être utilisés au travers du langage, nous ne considérons pas le langage comme étant capable de décrire de telles règles.

Contrairement à Snort, Bro et NeMode offrent des fonctionnalités permettant de décrire des règles mettant en relation plusieurs paquets réseaux. En effet, le langage de script Bro propose des librairies gérant les protocoles tels que TCP ou UDP. Des structures de données stockent les informations relatives à la connexion courante. Toutefois, ces structures ne peuvent pas être modifiées pour ajouter d'autres données. Concernant le langage NeMode, il est possible d'exprimer une signature pour laquelle plusieurs paquets sont en jeu. Le Listing 5.8 illustre cette fonctionnalité. Dans cet exemple, les paquets `A`, `B` et `C` sont utilisés conjointement pour détecter une tentative d'usurpation de réponse DNS.

Listing 5.8– Exemple de règle NeMode faisant intervenir plusieurs paquets

```
1  dns_spoofing {
2      udp_packet(A), dst_port(A) == 53,
3      udp_packet(B), src_port(B) == 53,
4      dst(B) == src(A), dst_port(B) == src_port(A),
5
6      udp_packet(C), src_port(C) == 53,
7      dst(C) == src(A), dst_port(C) == src_port(A),
8
9      B != C,
10     data(B,0,2) == data(A,0,2),
11     data(C,0,2) == data(A,0,2)
12 } => {
13     alert('DNS Spoofing attempt')
14 };
```

Bien que Bro et NeMode proposent des analyses sur des séquences de paquets, l'écriture de ces règles reste assez limitée. En effet, il n'est pas possible pour une règle Bro d'enrichir la structure de données utilisée. En ce qui concerne NeMode, il n'est pas possible d'étendre ces analyses sur des protocoles non supportés par le langage, car les protocoles sont figés dans le langage (comme l'indique la directive `udp_packet` par exemple). Nous développons dans les paragraphes suivants la question de l'extensibilité des langages.

Détection collaborative

Nous détaillons dans le paragraphe précédent les différents niveaux de granularité des langages. Outre l'analyse de la granularité du paquet ou de la séquence de paquets, la dernière granularité concerne l'aspect collaboratif du langage. Autrement dit, est-ce que le langage propose des moyens de faire collaborer plusieurs solutions de sécurité afin de détecter des intrusions.

Contrairement à Discus Script, les langages existants de la littérature ne proposent pas de fonctionnalités qui permettent de faire collaborer plusieurs solutions de sécurité. Ces langages se concentrent sur la configuration d'une seule cible, qui agit alors en un seul point du réseau.

Notre solution, Discus, propose une architecture massivement distribuée, constituée de solutions de sécurité qui collaborent au travers du mécanisme des tables. En effet, ces structures de données sont partagées sur le réseau et la stratégie de distribution est dépendante de la topologie réseau. Autrement dit, nos outils se chargent de répartir les tables sur le réseau en fonction des spécificités du réseau et des solutions de sécurité. Chacune des solutions, dont la configuration est générée spécifiquement, peut accéder à l'ensemble des tables et partager sa connaissance locale.

L'utilisation de cette fonctionnalité permet de détecter des attaques se produisant à large échelle. Nous montrons un cas d'utilisation dans la Section 4.4, où nous présentons plusieurs

règles Discus Script pour détecter les attaques SYN Flood. Dans cette configuration, l'ensemble des solutions de sécurité modifie la table associée à la détection de l'attaque. Si la collecte des événements était locale, cela ne prendrait en compte que les paquets réseaux traversant la solution de sécurité. Dans notre cas, l'ensemble des solutions met à jour la structure de données, ce qui accélère le processus de détection.

Extensibilité du langage

La sécurité réseau est un domaine d'étude qui change en permanence. Les vulnérabilités et faiblesses des systèmes, utilisées lors des intrusions, sont corrigées aussi rapidement que possible et les attaquants cherchent constamment de nouveaux points d'entrée. C'est pourquoi un certain degré d'extensibilité du langage est requis pour ce domaine. Nous entendons par extensibilité la capacité d'un langage à exprimer de nouveaux concepts, de nouvelles vulnérabilités.

En ce qui concerne Discus Script, nous avons adopté une approche minimaliste vis-à-vis de la construction du langage. Autrement dit, peu de concepts sont fixés dans notre langage, par exemple au travers de mots-clés. En effet, hormis l'événement élémentaire `packet`, l'ensemble des règles et concepts s'exprime grâce au langage, qu'il s'agisse des protocoles et de leur décomposition ou de la manipulation des tables. Cela implique que le support d'un nouveau protocole ne nécessite pas d'autres opérations que l'écriture de plusieurs règles décomposant les différents champs dudit protocole.

Par exemple, le Listing 5.9 illustre la règle qui est déclenchée lors de l'événement élémentaire `packet`. En considérant que le support utilisé est Ethernet, la règle manipule le `bitstream` afin d'en extraire les champs Ethernet.

Listing 5.9– Règle déclenchée lors de l'événement élémentaire packet

```
1 on packet(bitstream payload)
2     raise ethernet_packet(payload[48:95], payload[0:47], payload[96:111], payload[112:]);
3
4 on ethernet_packet(int48 mac_src, int48 mac_dst, int16 ethertype, bitstream payload)
5     where ethertype == 0x0800
6     raise ipv4_packet_raw(payload);
```

Cette extensibilité n'est pas accessible aussi facilement pour les autres langages, qui ont tendance à se reposer uniquement sur une base Ethernet et IP. Cela se remarque pour Snort, Bro, ainsi que NeMode, qui proposent dans le langage des mots-clés consacrés à UDP ou TCP. L'arrivée d'un nouveau protocole nécessiterait alors la modification du compilateur, ainsi que l'écriture de routines en C gérant spécifiquement le protocole en question.

Synthèse

Dans cette section, nous avons étudié l'expressivité de Discus Script, en le comparant avec plusieurs langages représentatifs de la littérature. Nous avons comparé ces langages sur plusieurs caractéristiques clés du domaine, telles que les mécanismes de détection exprimables ou l'extensibilité du langage. Notre langage se démarque des langages existants de l'état de l'art, en offrant plus de fonctionnalités. Le Tableau 5.1 illustre le gain apporté par Discus Script par rapport aux autres langages.

	Détection d'anomalies	Détection par signatures	Détection collaborative	Mécanisme de détection à états	Langage extensible
Discus Script		✓	✓	✓	✓
Snort		✓			
Bro		✓		✓	
NeMode		✓		✓	
Lambda		✓		✓	
STATL		✓		✓	
PLAN-P	✓				✓
binpac	✓				✓
Zebra	✓				✓

Table 5.1 – Tableau récapitulatif des propriétés des langages

5.2 Robustesse du langage

Dans la précédente section, nous avons étudié l'expressivité de Discus Script, en le comparant aux autres langages existants de l'état de l'art, selon certaines caractéristiques. Nous avons montré dans cette étude que notre langage est capable de proposer plus de fonctionnalités que d'autres langages. Dans cette section, la suite de l'évaluation de Discus Script s'intéresse à la robustesse du langage et de sa chaîne de compilation. Nous analysons dans cette section la capacité du langage à aider le développeur lorsque ce dernier commet une faute. Dans un premier temps, nous décrivons le cas d'utilisation visé par cette étude, puis nous présentons le protocole

expérimental et les résultats associés.

5.2.1 Outils liés au langage

Un langage dédié est accompagné par une suite d'outils, comme par exemple l'analyseur syntaxique qui se charge de la lecture du fichier source pour créer les unités lexicales (ou *tokens* en anglais). Quant à l'analyseur grammatical, il s'assure que les tokens forment des phrases respectant la syntaxe fixée par le langage. Pour finir, l'analyseur sémantique s'assure que les phrases du langage ont du sens, en fonction du domaine d'étude. Ces trois outils principaux forment la base d'un langage et permettent de raisonner sur le langage source.

En ce qui concerne Discus Script, en plus de ces outils de base, nous proposons un ensemble d'outils qui permet d'affiner l'analyse et l'interprétation des fichiers sources. Parmi ces outils, nous pouvons retrouver :

— les **outils d'analyse**, qui se chargent de l'analyse d'un fichier source, en vue de mettre en avant des propriétés relatives au domaine ;
— les **outils d'optimisation**, qui se basent sur une analyse préalable et manipulent le fichier source pour enrichir son contenu ;
— les **outils de traduction**, qui génèrent le fichier binaire ou source correspondant à la solution de sécurité à déployer.

L'ensemble de ces outils permet de manipuler un fichier source et de vérifier que son contenu respecte les concepts du domaine étudié. L'utilisation de ces outils apporte des propriétés de robustesse au langage, que nous détaillons dans la prochaine section. Notamment, nous avons développé des outils permettant de :

— vérifier que l'ensemble des éléments est préalablement défini ;
— vérifier que le graphe d'événements ne contient pas d'événements non atteignables, pour un événement initial donné ;
— supprimer les séquences d'événements ne menant pas à une action ou qui ne seraient pas applicables sur la topologie réseau où nous souhaitons déployer nos solutions de sécurité ;
— supprimer les tables qui ne sont pas utilisées dans une configuration réseau donnée ;
— vérifier que les éléments réseau sont valides ;
— détecter des cycles d'événements, afin de s'assurer de la terminaison de l'analyse d'un paquet réseau.

5.2.2 Mutation de code source

Dans cette section, nous cherchons à montrer que notre langage est aussi robuste que les autres langages existants, voire meilleur. Nous entendons par robustesse la capacité du langage

à détecter une incohérence dans le fichier source étudié, quelque soit l'outil utilisé de la chaîne de compilation. Une incohérence peut être assimilée à une erreur dans le langage, qu'il s'agisse d'une erreur lexicale, grammaticale ou sémantique.

Différents types de mutations

Pour pouvoir évaluer la robustesse de notre langage, nous avons élaboré un banc d'expérimentations simulant des erreurs de saisie, qui seraient réalisées par le développeur en charge de l'écriture de la politique de sécurité. Nous avons identifié plusieurs erreurs de saisie pour notre étude, présentées dans les prochains paragraphes. Ces erreurs simulées permettent de générer des fichiers sources corrompus, qui seront analysés par la chaîne de compilation, qui conclue si le fichier en question est sain ou non. Nous considérons dans cette étude qu'un langage robuste est un langage capable de détecter un grand nombre d'erreurs.

Modification d'un mot du langage La première modification considérée est une erreur de saisie commune, relative aux mots du langage. Un mot du langage est une unité lexicale, c'est-à-dire un segment du code source identifié par l'analyseur lexical, aussi appelé *token* en anglais. Les mots du langage peuvent être sujets à de nombreuses modifications, notamment la **suppression**, l'**ajout**, la **substitution** ou l'**inversion** de caractères.

Par exemple, le mot du langage `tcp_packet` peut être modifié en `tcp_apcket` en inversant deux lettres voisines. La détection de ce type de mutation peut être facile sur certains types de mots du langage, comme les mots-clés. Cependant, la détection est plus ardue quand il s'agit de valeurs entières, ou d'adresses réseaux.

Modification d'une valeur numérique ou flottante Les valeurs numériques ou flottantes du langage sont sujettes aux mêmes types de mutations. Dans notre étude, nous avons soumis ces valeurs à des **substitutions**, **suppressions** ou **insertions** de chiffres.

Par exemple, la valeur `443` pourrait être remplacée par `4439`. La détection de ce type de mutation n'est pas facile, car la mutation ne provoque pas d'erreur lexicale ou grammaticale. Il est nécessaire de se reposer sur la sémantique pour détecter ces mutations, par exemple en s'assurant qu'une adresse IPv4 n'est pas constituée d'octet valant plus que 255.

Substitution de mots du langage Les deux premières catégories de mutations se concentrent sur des erreurs de saisie communes, souvent dues à l'inattention du développeur. Nous visons dans cette catégorie les erreurs liées aux fonctionnalités de complion automatique, souvent intégrées dans les éditeurs de texte modernes. Ces fonctionnalités visent à faciliter le développement, en limitant la quantité d'information à saisir et en proposant des compléments qui correspondent à la chaîne de caractères que le développeur a commencé à taper. Par exemple, si l'utilisateur commence à taper `tcp_`, son éditeur pourrait lui proposer la complétion `tcp_packet`, puisque ce

mot du langage est utilisé dans une autre section du fichier source. Ce type de fonctionnalité peut engendrer des fautes dans le fichier source, à l'insu du développeur, en choisissant des complétions inadaptées ou incorrectes.

Nous appliquons dans notre étude des mutations de ce type sur les identifiants (nom de fonction, de variable, mot-clé du langage, etc.) reproduisant ce phénomène. La mutation consiste à remplacer un identifiant par un autre, dans le but de simuler une complétion incorrecte. Ce type de mutation peut être détecté grâce à l'analyse grammaticale ou sémantique, mais des cas subsistent où la substitution semble légitime.

Mutation des unités lexiales du langage

Les mutations jusque là présentées ne sont pas applicables pour l'ensemble des tokens du langage. En effet, la modification d'une valeur numérique est spécifique à un certain type de mots du langage. Pour notre étude, nous avons distingué différents types de mots du langage :

- une **valeur numérique entière** est une constante entière dans le code source, telle que `8080` ou `127.0.0.1` (constituée de 4 valeurs entières) ;
- une **valeur entière hexadécimale** est également une constante dans le code source, telle que `0xDEADBEEF` ;
- une **chaîne de caractère** est une constante dans le code source, bornée par des guillemets, telle que `"GET /.*"` ;
- une **déclaration de variable** est la première occurrence d'une variable dans le code source, où cette dernière est déclarée ;
- une **déclaration de fonction** correspond à l'identifiant d'une fonction lorsque celle-ci est déclarée (dans le cas de Discus Script, nous considérons qu'un événement est une fonction) ;
- une **référence** est un identifiant qui se rapporte à une variable ou une fonction précédemment déclarée ;
- un **mot-clé du langage** est une unité lexicale fixée dans la syntaxe du langage, tel que `on` ou encore `where`.

Nous présentons dans le Tableau 5.2 les mutations possibles pour chacun de ces mots du langage.

5.2.3 Expérimentations

Dans cette section, nous présentons les expérimentations réalisées, pour valider la robustesse de Discus Script. Dans un premier temps, nous détaillons le protocole expérimental, ensuite nous présentons les fichiers sources étudiés et pour finir les résultats obtenus au cours de ces analyses.

	Inversion de caractères	Suppression de caractères	Substitution de chiffres	Insertion de chiffres	Substitution de mots
Valeur numérique entière	✓	✓	✓	✓	
Valeur hexadécimale	✓	✓	✓	✓	
Chaîne de caractères	✓	✓			
Déclaration de variable	✓	✓			✓
Déclaration de fonction	✓	✓			✓
Référence	✓	✓			✓
Mot-clé du langage	✓	✓			✓

Table 5.2 – Tableau récapitulatif précisant les mutations possibles pour chaque type de mots du langage

Protocole expérimental

Pour mener cette série d'analyses, nous avons établi un protocole expérimental, découpé de la manière suivante :

1. choisir un fichier source sain du langage à étudier ;
2. opérer une mutation sur le fichier source en question, pour un type de mots du langage donné ;
3. utiliser la chaîne de compilation du langage pour analyser le code source modifié ;
4. étudier le comportement de la chaîne de compilation et conclure si la mutation a été détectée ou non.

Pour un langage et un type de mot du langage donné, cette séquence d'actions décrit une itération du protocole expérimental que nous avons suivi pour cette étude. Nous avons choisi de n'opérer qu'une seule mutation par itération, pour avoir des résultats précis en ce qui concerne les conséquences d'une mutation. En effet, certaines mutations peuvent produire un code légitime aux yeux de la chaîne de compilation et la démultiplication de mutations aurait favorisé la détection d'erreurs.

Pour chacun des types de mots du langage (valeur numérique entière, hexadécimale, etc.), nous avons réalisé 1000 itérations successives, ce qui nous permet d'avoir des chiffres précis. À

cela, nous avons ajouté 1000 itérations où la mutation choisit aléatoirement un mot du langage dans le code source, afin de proposer une étude représentative (en terme de type de mots du langage) du code source examiné.

Fichiers sources étudiés

Pour nos expérimentations, nous avons étudié la robustesse de Discus Script ainsi que celle du langage de Snort. Cette dernière étant adoptée massivement par la communauté, elle nous a semblé être la solution idéale pour une étude comparative.

Pour comparer des fichiers sources équivalents, nous avons choisi un fichier source issu de la communauté Snort, décrivant plus de 600 signatures de sécurité. Nous avons ensuite traduit ce fichier, grâce à l'outil présenté dans le début de ce chapitre, pour obtenir un fichier source Discus Script. Les deux fichiers correspondent donc aux fichiers sains initiaux et seront sujets à des mutations.

	Discus Script	Snort
Valeurs entières et hexadécimales	1243	2772
Chaîne de caractères	1408	1325
Déclaration de variables	2007	8
Déclaration de fonctions	627	-
Références	12481	1506
Mot-clés du langage	5150	9645
Total	22916	15256

Table 5.3 – Composition des fichiers sources étudiés en fonction des types de mots du langage

Nous présentons dans le Tableau 5.3 la composition de ces deux fichiers, en fonction des différents types de mots du langage. Pour la même quantité d'information, un fichier Discus Script est environ `1.5` fois plus verbeux qu'un fichier Snort, en ce qui concerne le nombre de mots du langage des fichiers sources. Ce n'est pas surprenant puisque la construction du langage se veut plus verbeuse dans Discus Script.

La traduction du fichier Snort vers Discus Script ne prend en compte qu'une partie des informations ; notamment, l'ensemble des méta-données n'est pas traduit, composées en majeure partie d'entiers et de chaînes de caractères. Ceci explique le nombre moins important de valeurs entières dans le fichier Discus Script. Le langage Snort ne propose pas de notion de fonction et les variables (globales) sont déclarées pour identifier les réseaux et groupes de ports. Dans le cas de Discus Script, nous considérons qu'un événement est assimilable à une fonction et les variables sont majoritairement des arguments de fonctions. Les mots clés sont moins nombreux dans notre langage, dû à notre volonté de pouvoir exprimer le maximum de concepts en utilisant uniquement notre langage.

Résultats

Après avoir appliqué le protocole expérimental décrit précédemment, nous avons calculé le taux de détection de mutations, décrit par la formule suivante :

$$\text{taux de détection} = \frac{\text{nombre de mutations détectées}}{\text{nombre total de mutations}}$$

Le Tableau 5.4 présente les taux de détection pour chacun des types de mots des langages. Plus le taux de détection est élevé, plus le langage est robuste car capable de détecter un grand nombre de mutations du code source. Ce tableau peut être lu de la manière suivante : si la mutation est appliquée sur une valeur entière ou hexadécimale, Discus Script est capable de détecter 9.50% de mutations alors que Snort ne peut en détecter que 5.90%.

	Discus Script	Snort
Valeurs entières et hexadécimales	9.50	5.90
Chaîne de caractères	27.00	22.00
Déclaration de variables	64.30	100.00
Déclaration de fonctions	100.00	-
Références	99.30	76.60
Mot-clés du langage	99.90	97.60
Tous types de mots du langage	87.05	73.40

Table 5.4 – Taux de détection (en pourcentage) des mutations selon les types de mots du langage

Globalement, Discus Script est plus robuste que Snort, quand la mutation choisit aléatoirement un mot du langage. En effet, notre langage détecte 87% des mutations alors que Snort n'en détecte que 73%. En ce qui concerne les mutations propres à chaque type de mots du langage, Discus Script est plus robuste que Snort, excepté pour les déclarations de variables.

Pour le cas des déclarations de variables, le fichier source Discus Script est composé de 2007 déclarations, contre 8 pour Snort. Parmi ces 2007 déclarations de variables, il s'agit principalement des paramètres des événements. Le Listing 5.10 propose un exemple de mutation. La mutation en question a transformé `seq_num` en `eqs_num`. Dans un tel cas de figure, la mutation ne peut être détectée que si la variable est utilisée par la suite. Si cette variable n'est pas utilisée, alors la déclaration de la variable `eqs_num` est aussi légitime que celle de `seq_num`. Par conséquent, le compilateur ne peut en conclure qu'une mutation a eu lieu. Les mutations de déclaration de variables qui ne sont pas détectées par le compilateur correspondent intégralement à ce cas de figure. Dans le code Discus Script, les événements principaux ont un grand nombre de paramètres, tels que `tcp_packet` qui a 12 paramètres. Parmi ces paramètres, peu sont utilisés dans toutes les règles, c'est pourquoi un certain nombre de mutations sur les paramètres n'est pas détecté. La mutation n'est pas détectée, mais cela n'a aucune incidence sur le fichier source.

Listing 5.10– Exemple de mutation d'une déclaration de variable

```
1  /* Initial rule */
2  on tcp_packet(ipaddr src, ipaddr dst, int16 src_port, int16 dst_port, int32 seq_num, [...])
3      where (src in SQL_SERVERS)
4          [...]
5      ;
6  /* Mutated rule */
7  on tcp_packet(ipaddr src, ipaddr dst, int16 src_port, int16 dst_port, int32 eqs_num, [...])
8      where (src in SQL_SERVERS)
9          [...]
10     ;
```

Pour les mutations sur les autres types de mots du langage, Discus Script est plus robuste que Snort. Notre langage est capable de réaliser les détections en combinant l'analyse lexicale, grammaticale et sémantique. Conjugués à ces outils, des analyseurs vérifient que le graphe d'événements est cohérent et ne provoque pas de cycles. Pour finir, le typage de notre langage nous permet de vérifier plus facilement qu'une expression est correcte, puisque l'on vérifie que le type de l'expression en question est bien celui attendu.

En ce qui concerne la mutation de valeurs numériques, Discus Script et Snort sont tous deux incapables de détecter efficacement ces mutations, qui réalisent respectivement des taux de 9.50% et 5.90% des mutations. Les mutations réalisées (présentées dans le Tableau 5.2) produisent pour la majorité des cas un résultat qui parait légitime. En effet, nos outils ne peuvent détecter la mutation de la valeur `8080` en `8008`. Toutefois, des analyses nous permettent d'être plus robuste que Snort, par exemple en vérifiant que les IP du fichier source sont valides.

5.2.4 Synthèse

Dans cette section, nous nous sommes intéressé à la robustesse de Discus Script, c'est à dire sa capacité à détecter qu'une inconsistance est présente dans le fichier source étudié. Pour cela, nous avons mené des expérimentations qui ont simulé des erreurs de saisie courantes et nous avons mesuré le taux de détection des mutations. Nous avons comparé la robustesse de Discus Script au langage de Snort, une solution de l'état de l'art populaire et communément admise comme solution de choix. Notre solution se révèle plus robuste dans l'ensemble des cas d'étude et est capable de détecter 87% des mutations contre 73% pour Snort. Les outils accompagnant le compilateur permettent de déceler plus facilement les incohérences. Les choix réalisés lors de la phase de conception permettent de faciliter la détection de ces erreurs, notamment le typage statique des expressions.

Nous avons évalué dans cette section le langage et ses propriétés de robustesse. Nous nous intéressons dans la prochaine section à d'autres métriques pour évaluer notre solution, plus précisément les performances de la solution de sécurité générée par la chaîne de compilation

Discus Script.

5.3 Performances

Nous avons précédemment étudié l'expressivité et la robustesse de Discus Script, un langage dédié à la description de règles de sécurité. Ces premières études se concentraient sur le langage et ses propriétés. L'élaboration d'un langage dédié n'est que la première phase de notre solution. En effet, de cette abstraction, nous souhaitons générer des fichiers pour configurer l'ensemble des systèmes de sécurité. La dernière étude, séparée en trois aspects, se concentre sur l'évaluation du système généré par notre chaîne de compilation.

Nous avons conduit notre étude de ce système selon trois mesures de performances : l'empreinte mémoire du système, son efficacité en terme de taux de détection d'intrusions et pour finir le temps de traitement d'un paquet. Ces études s'intéressent donc à la fois à des aspects système, réseau et sécurité. Avant de nous intéresser à ces études, nous présentons dans un premier temps la configuration déployée pour évaluer ces différentes mesures, communes à l'ensemble des expérimentations.

5.3.1 Configuration du banc d'essai

Nous présentons dans cette section la configuration du banc d'essai mis en place pour l'ensemble des études s'intéressant aux performances de notre solution. Les trois études proposées examinent l'impact et l'efficacité du système de sécurité généré, en terme de mémoire, de détection d'intrusions et de rapidité de traitement. Afin d'avoir un socle commun pour ces études, les mesures ont été réalisées sur le même banc d'essai.

Nos travaux se sont concentrés sur la configuration des solutions de sécurité, au travers d'un langage dédié, exprimant des règles de sécurité basées sur une approche événementielle. Dans ces travaux, nous avons réalisé une chaîne de compilation dont l'objectif est de générer le code associé à chacune des solutions de sécurité prenant part au système global. Puisque nos travaux ne se concentrent pas sur l'aspect distribué de Discus, nous n'avons pas mis en place une structure d'expérimentations distribuée.

Le banc d'essai est constitué de trois machines physiques, qui joueront respectivement le rôle d'attaquant, de solution de sécurité et de cible. La machine attaquante envoie du trafic vers la cible, au travers de la solution de sécurité, qui analyse les paquets réseaux et en conclut si une attaque est en cours ou non.

Pour faciliter la reproductibilité de l'expérimentation, nous avons utilisé un fichier de trace réseau public, délivré par l'outil Tcpreplay [93], utilisé par la machine attaquante pour générer le trafic en question. Le fichier de trace [1] correspond au trafic réseau d'une passerelle d'un réseau

1. http ://tcpreplay.appneta.com/wiki/captures.html (smallFlows.pcap)

privé d'entreprise, pour une durée totale de 5 minutes et un peu plus de 14.000 paquets réseaux. Le trafic a été modifié afin de correspondre aux plages d'adressages du réseau attaquant et attaqué.

Les mesures sont réalisées au niveau de la solution de sécurité. Nous avons instrumenté le code des solutions testées pour mesurer la consommation mémoire, le taux de détection et la durée de traitement d'un paquet réseau. Le rôle de la solution de sécurité est de détecter les intrusions et de rester passive le cas échéant.

Pour ces trois études, nous avons comparé les performances de la solution générée par Discus Script et Snort. Ces deux solutions ont été instrumentées pour faciliter la prise de mesures, notamment pour examiner finement leur consommation mémoire et le temps d'analyse moyen d'un paquet. Pour comparer équitablement ces deux solutions, nous avons utilisé les mêmes signatures de détection d'intrusions, précédemment utilisées pour l'analyse de la robustesse du langage.

5.3.2 Empreinte mémoire

Présentation de la métrique

La première étude se concentre sur l'empreinte mémoire des solutions. Cette évaluation est intéressante, car notre étude porte sur un système massivement distribué, constitué de solutions hétérogènes. Certaines d'entre elles auront des capacités limitées en terme de mémoire, c'est pourquoi il est nécessaire de proposer une solution peu gourmande en espace mémoire.

Dans le cas d'une solution générée par Discus Script, la mémoire est principalement utilisée au travers des tables, qui se remplissent au fur et à mesure de l'analyse, comme les tables qui renseignent l'état des connexions TCP, ou encore celles qui dénombrent les événements suspects. Également, les paramètres des événements sont alloués dynamiquement. Les instructions `purge` et `delete` de Discus Script permettent de gérer les mécanismes de libération de la mémoire dynamiquement allouée, pour éviter des famines mémoire. D'autre part, les paquets réseaux sont placés dans des espaces statiquement alloués lors de leur analyse. En ce qui concerne Snort, la gestion de la mémoire est réalisée de manière similaire, mais ne propose pas d'opérations spéciales en cas de famine. Tout comme notre solution, une phase d'initialisation prépare Snort, au cours de laquelle l'ensemble des expressions régulières est compilé et chargé.

Pour cette expérimentation, nous avons modifié les deux solutions étudiées de manière à ce que la gestion de la mémoire dynamique, c'est-à-dire l'ensemble des fonctions manipulant la mémoire telles que `malloc`, `free` ou `calloc`, soit réalisée par un outil d'analyse mémoire. Ce dernier intercepte les appels à ces fonctions, appelle les fonctions adéquates et enregistre la consommation mémoire courante du programme. Pour simuler un environnement contraint en mémoire, cet outil d'analyse limite l'utilisation de la mémoire dynamique à 1 mégaoctet. Bien que cette quantité ne représente pas nécessairement les environnements contraints ciblés, cette

quantité permet de rencontrer des états de famine pour la trace réseau étudiée.

Résultats

Les Figures 5.1 et 5.2 représentent l'évolution de l'empreinte mémoire de Snort et Discus Script en fonction du temps, respectivement pour 200 et 400 signatures. Plus la consommation mémoire est haute, plus le programme a réalisé d'allocations dynamiques. L'outil d'analyse de la mémoire, que nous avons intégré aux deux solutions, limite la consommation à 1 mégaoctet, pour simuler un environnement contraint en mémoire. Cette limite est représentée par une ligne horizontale sur les graphes.

La consommation mémoire de Discus Script réside dans l'initialisation des expressions régulières, les tables et les paramètres des événements. Seule la consommation des tables est vouée à croître indéfiniment, jusqu'au moment où la limite mémoire est atteinte et le processus de purge enclenché.

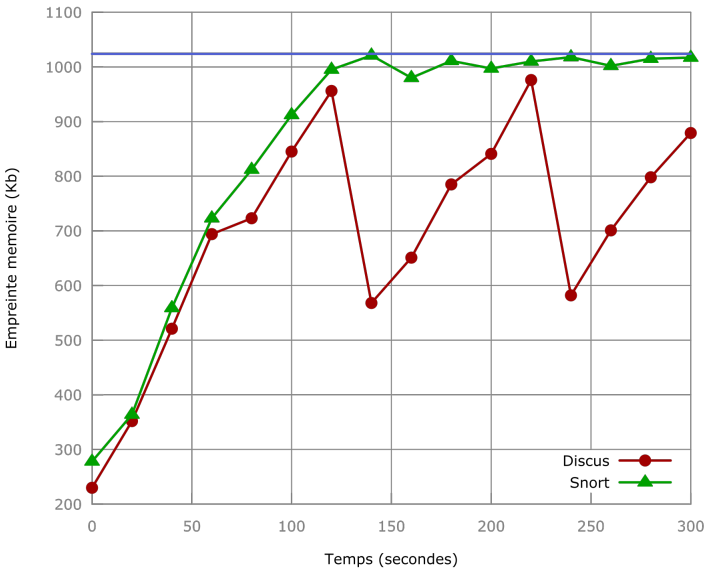


Figure 5.1 – Évolution de l'empreinte mémoire de Snort et Discus Script en fonction du temps (200 signatures)

L'empreinte mémoire initiale de Discus Script est plus faible en début d'expérimentations. La nature spécifique de notre solution pour cet ensemble de règles permet d'allouer peu de mémoire dynamique à l'initialisation. Ensuite, la consommation mémoire de Discus Script adopte une courbe équivalente à celle de Snort (bien que généralement en deçà), jusqu'au moment où la limite mémoire est atteinte par chacune des solutions. Le comportement de Discus Script est alors d'invoquer la fonction de purge des tables, qui va libérer de l'espace mémoire. Quant à Snort, une fois la mémoire entièrement occupée, aucun traitement de purge n'est effectué. Il continue à réaliser des allocations dynamiques, qui échouent pour la plupart si aucune libération

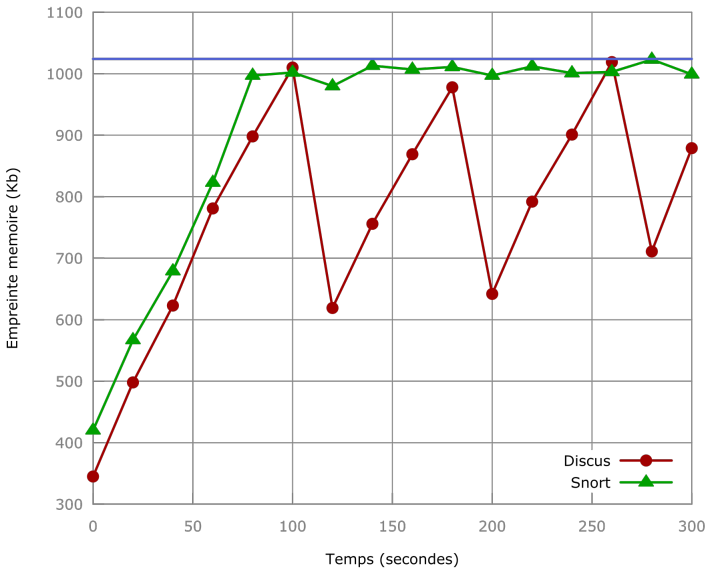


FIGURE 5.2 – Évolution de l'empreinte mémoire de Snort et DISCUS SCRIPT en fonction du temps (400 signatures)

n'a été réalisée.

Cet environnement contraint produit deux comportements bien distincts sur les solutions à l'étude. En ce qui concerne DISCUS SCRIPT, nous décidons sciemment de libérer de l'espace en choisissant les enregistrements à supprimer. Snort ne change pas de méthode quand il atteint la limite mémoire, il poursuit ses tentatives d'allocations mémoire, qui échouent pour la plupart, sauf si des données précédemment allouées ont été libérées. Pour conclure, la différence entre ces deux solutions est donc que Snort ne peut plus stocker les données relatives aux nouvelles connexions, alors que DISCUS SCRIPT choisit de supprimer les données les plus anciennes.

5.3.3 Temps de traitement d'un paquet

Présentation de la métrique

Dans la première étude, nous nous sommes intéressés à la consommation mémoire de la solution de sécurité et nous avons pu constater que DISCUS SCRIPT consomme moins de mémoire et la gestion de celle-ci s'adapte aux cibles visés, en libérant de la mémoire quand cela est nécessaire. Dans cette section, nous étudions le temps de traitement d'un paquet par la solution de sécurité, c'est-à-dire la durée moyenne de l'analyse complète d'un paquet réseau.

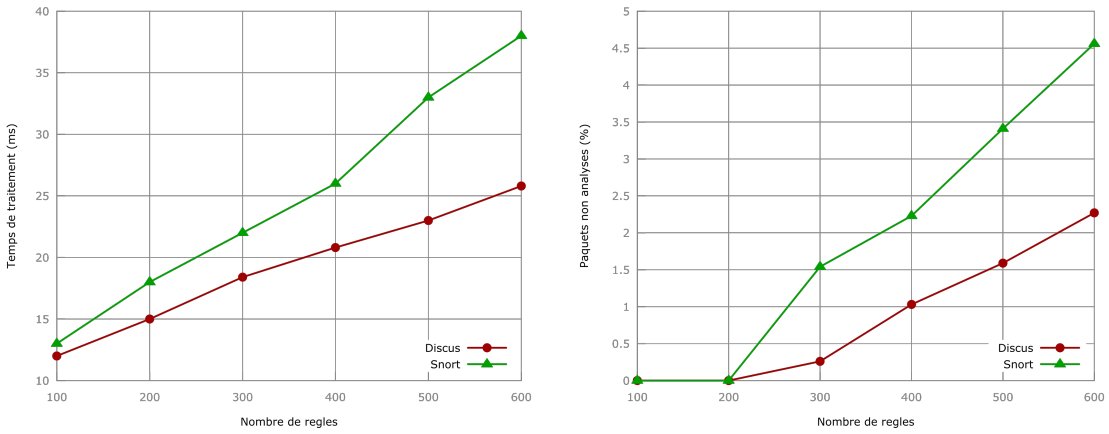
Il est primordial pour de tels programmes de proposer un temps de traitement faible, pour supporter la charge réseau et traiter l'ensemble du trafic. Pour cette série d'expérimentations, notre banc d'essai reste inchangé et les solutions sont limitées en espace mémoire, tel que précisé dans la section précédente. Pour mesurer la durée moyenne d'analyse d'un paquet, nous avons

modifié le traitement d'un paquet des solutions, pour pouvoir mémoriser la date d'arrivée d'un paquet et la fin de son analyse, qui seront ensuite analysées une fois l'expérience terminée.

En plus de cette métrique, nous nous proposons d'étudier le taux de paquets non traités par la solution. Dans le cas où la durée d'analyse moyenne des paquets est trop élevée, la solution ne peut pas réceptionner l'ensemble des paquets arrivants. Des mécanismes de file d'attente sont mis en place dans les deux solutions, mais ne peuvent pas résoudre ce problème, car ces files seront ultimement remplies. Puisque nos solutions sont des IDS et non des IPS, le trafic entrant est directement routé vers le port réseau de sortie. Par conséquent, l'acheminement du trafic est effectif et il n'y a pas de paquet perdu. Cependant, il est probable que la solution de sécurité ne puisse analyser l'ensemble des paquets et de ce fait ne détecte pas certaines attaques.

Résultats

La Figure 5.3 présente les résultats de cette étude. Le graphe représente la durée de traitement moyenne (en millisecondes) par rapport au nombre de signatures. Plus la durée de traitement est faible, plus la solution sera capable d'analyser de paquets par seconde. Également, plus le taux de paquets non traités est faible, plus la solution étudie l'intégralité du trafic réseau.



(a) Durée de traitement d'un paquet (b) Taux de paquets non traités

Figure 5.3 – Durée de traitement moyenne par paquet pour Snort et Discus Script en fonction du nombre de règles

Quelque soit le nombre de signatures, la durée de traitement moyenne d'un paquet est toujours supérieure pour Snort. Cela signifie qu'il faut davantage de temps à Snort pour analyser les paquets, pour une base de signatures équivalente. Pour la solution Snort, la base de signatures est constituée de règles qu'il faut évaluer les unes après les autres. C'est pourquoi la durée moyenne de traitement adopte une courbe presque linéaire. En ce qui concerne Discus Script, seules les règles concernées sont déclenchées, ce qui implique qu'une sous-partie de la politique de sécurité ne sera pas évaluée. Ceci explique pourquoi cette durée est moins élevée pour notre solution.

D'autre part, dès 300 signatures, on peut constater que des paquets ne sont plus traités, quelque soit la solution. Cela signifie que pour ce palier, des paquets réseaux ont été rejetés par la solution de sécurité, déjà occupée à analyser des paquets précédents. Toutefois, il faut remarquer que la perte est moins élevée pour Discus Script, qui prend moins de temps en moyenne pour analyser les paquets réseaux. Cela peut avoir des conséquences sur le taux de détection, car l'ensemble des paquets n'a pas été analysé. Nous nous intéressons à cela dans la prochaine section.

5.3.4 Taux de détection

Présentation de la métrique

La dernière mesure d'évaluation s'intéresse à une métrique du domaine de la sécurité, le taux de détection d'intrusions. Il est exprimé de la manière suivante :

$$T = \frac{\text{nombre d'intrusions détectées}}{\text{nombre total d'intrusions}}$$

Cette métrique permet d'évaluer la précision d'une solution de sécurité et sa capacité à détecter l'ensemble des événements malveillants. Dans notre cas, les intrusions que nous pouvons détecter sont décrites sous forme de signatures, c'est pourquoi nous considérons que le nombre total d'intrusions ne comprend que les attaques correspondant à ces signatures. Il est donc possible que certaines intrusions ne soient pas détectées, puisqu'aucune signature n'existe pour ces intrusions. Cependant, comme notre étude s'intéresse à l'efficacité du langage Discus Script et de la solution générée, cette métrique sera idéale pour se comparer à Snort, pour un ensemble de signatures équivalent.

Notre série d'expérimentations se déroulera sur le banc d'essai précédemment présenté, avec les mêmes contraintes mémoires. Puisque ces contraintes peuvent interférer sur le taux de détection, nous avons réalisé tout d'abord une expérience visant à établir une base de référence. Pour cette expérience, nous avons mesuré le taux de détection de Snort, sans contrainte mémoire, qui constituera la référence en terme de taux de détection. La métrique que nous étudierons dans cette section consiste à se comparer à ce taux de référence, exprimé en pourcentage de la manière suivante :

$$R(n) = \frac{T(n)}{T_{\text{référence}}(n)} * 100$$

Nous étudierons donc le taux de détection relatif à un taux de référence, selon un nombre `n` de règles de sécurité.

Résultats

Nous présentons dans la Figure 5.4 le taux de détection relatif (en pourcentage) en fonction du nombre de signatures, de Snort et de Discus Script. Dans ce graphe, plus le taux est élevé, plus la solution constate l'intégralité des détections de référence. Autrement dit, un taux de détection relatif de 100% indique que la solution a perçu autant de détections que la solution de référence, qui n'est pas sujette à des contraintes mémoires.

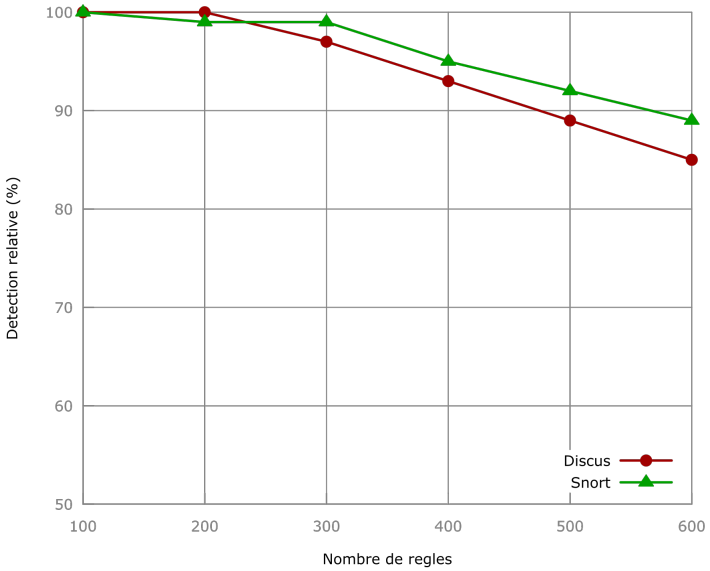


Figure 5.4 – Taux de détection relatif pour Snort et Discus Script

Lorsque peu de signatures sont utilisées par les solutions de sécurité, le taux de détection relatif est proche des 100%. Dès 300 signatures, le taux de détection décroît pour les deux solutions qui finissent en dessous de 90% de taux de détection relatif. Plusieurs facteurs peuvent expliquer cette baisse d'efficacité. Premièrement, l'environnement contraint en terme de mémoire provoque des pertes de données, comme nous l'avons vu dans la première série d'expérimentations. Cette perte de données, qui survient lorsque la mémoire ne permet plus d'allouer dynamiquement de l'espace, a pour conséquence que la solution ne peut plus stocker les informations relatives aux analyses des paquets. Deuxièmement, le taux de perte de paquets, étudié précédemment, indique que plus il y a de signatures, plus le nombre de paquets non traités est grand. D'ailleurs, ce phénomène se présente dès 300 signatures, valeur pour laquelle les premières pertes ont été constatées. Ceci corrobore le fait que certains paquets, indispensables à la détection d'intrusions, n'ont pas été analysés par les solutions de sécurité.

En ce qui concerne l'efficacité des solutions dans le même environnement, Discus Script est légèrement en deçà de Snort, de quelques pourcents. Cela peut être dû à la perte d'informations contextuelles au niveau de la mémoire ou à la perte de paquets, mais comme nous l'avons remarqué dans la précédente étude, le taux de paquets non traités est moins élevé pour notre solution. Plus le nombre de signatures est grand, plus la mémoire est sollicitée et plus les opérations de

purges sont invoquées. Nous avons remarqué d'importantes baisses de la consommation mémoire à ces périodes, provoquant la perte de nombreuses informations. Nous pensons alors que la phase de purge a été trop peu sélective et qu'elle a supprimé trop d'informations, amenant à la non détection de certaines intrusions. Nous sommes convaincus que les résultats de Discus Script seraient au moins équivalents à Snort, voire meilleurs, pour un nombre de signatures égal, à la simple modification des conditions de purge.

5.4 Synthèse

Dans ce chapitre, nous avons évalué Discus Script selon différents aspect, afin de valider notre langage. Nous avons tout d'abord étudié l'expressivité du langage, puis sa robustesse et enfin ses performances.

Dans un premier temps, nous avons étudié l'expressivité du langage, pour s'assurer qu'il est au moins aussi expressif que les langages existants de l'état de l'art. Pour cela, nous avons développé un outil permettant de traduire des règles Snort en règles Discus Script. Cela nous a permis de valider que notre langage est effectivement aussi expressif que Snort, mais également d'avoir un ensemble de règles équivalent pour les prochaines études. Nous avons poursuivi cette étude en comparant les notions que chacun des langages peut exprimer. De cette manière, nous avons mis en lumière que Discus Script est capable d'exprimer plus de concepts que les autres langages, tels que la collaboration ou l'analyse de séquence de paquets réseaux. D'autre part, notre langage est facilement extensible. Nous entendons par extensibilité d'un langage la capacité de celui-ci à pouvoir supporter de nouveaux protocoles réseau, sans modifier la chaîne de compilation.

Ensuite, nous nous sommes intéressés à la robustesse de Discus Script. Nous considérons qu'un langage robuste est capable de détecter les erreurs d'un fichier source et peut aider l'utilisateur à le corriger. Pour cela, nous avons comparé notre langage à Snort, en réalisant de nombreuses mutations de fichiers sources et en étudiant le comportement de ces deux langages. Ces mutations visent à simuler des erreurs de saisie courantes, telles des insertions, des suppressions et des substitutions de caractères. Cette série d'expérimentations a montré que Discus Script est plus robuste que Snort, car notre langage détecte en moyenne 87% des mutations, contre 73% pour Snort.

Pour finir, nous avons évalué les performances de Discus Script, en comparant la solution générée par notre langage à Snort. Nos études se sont intéressées à différentes métriques, telles que l'empreinte mémoire, la durée de traitement par paquet ou encore le taux de détection d'intrusions. Pour un ensemble de règles équivalent, notre solution est moins consommatrice en mémoire et surtout propose un système de purge efficace pour libérer de la mémoire. L'approche événementielle permet d'éviter le parcours séquentiel des règles de la politique de sécurité, impliquant un temps de traitement plus faible pour notre solution que pour Snort. Quant aux taux de détection des attaques, notre solution est légèrement moins efficace, mais nous pensons que notre politique de purge était trop agressive et provoquait la perte de données inutile.

6

Conclusion et perspectives

6.1 Synthèse

Dans ces travaux de thèse, nous nous sommes intéressés à la sécurité d'architectures réseaux complexes, telles que le Cloud Computing, composées de nombreux équipements. La sécurisation de ce système nécessite la surveillance globale du trafic, afin de détecter l'ensemble des attaques perpétrées, y compris les attaques internes à la structure. Les solutions de sécurité traditionnelles réalisent généralement leurs analyses en ne considérant que le trafic qui transite localement. De même, les traitements réalisés par ces équipements estiment que les attaques proviennent exclusivement de l'extérieur du réseau. Cette vision n'est malheureusement plus applicable pour des réseaux comme le Cloud Computing, où un utilisateur malicieux peut facilement obtenir des ressources puissantes, et à bas coût, pour réaliser des attaques provenant du réseau interne.

Pour palier à ces problèmes, nous proposons Discus, une architecture qui se repose sur les solutions de sécurité usuelles, ainsi que sur un réseau massivement distribué de sondes de sécurité, éparpillées sur le réseau afin de conduire des analyses fines du trafic. Pour surveiller l'ensemble des machines du parc, nous proposons deux types de sondes : les sondes physiques, placées en coupure sur le réseau, et les sondes virtuelles, intégrées aux machines physiques à surveiller. Pour détecter les attaques à large échelle, les sondes collaborent entre elles et s'échangent des informations. Plusieurs problématiques sont soulevées par notre approche, dont la configuration d'un parc aussi massif de systèmes sur laquelle nos travaux se concentrent.

La configuration de l'ensemble de ces sondes nécessite la maîtrise de nombreux domaines, tels que la sécurité, la programmation embarquée ou encore la programmation distribuée. D'autre part, l'hétérogénéité des sondes implique la nécessité de configurer individuellement chacune des solutions de sécurité. Pour ces raisons, il n'est donc pas envisageable de confier la configuration à un petit groupe de personnes. Nous proposons dans ces travaux le langage dédié Discus Script, qui fournit une couche d'abstraction permettant de se concentrer sur la description d'une politique de sécurité plutôt que sur des aspects très spécifiques au développement des cibles déployées.

6.2 Résumé des contributions et publications

Nous résumons dans cette section les contributions de la thèse en faisant référence aux publications qui en découlent, détaillées dans les pages suivantes.

Dans un premier temps, nous nous sommes intéressés à l'impact des attaques sur des structures similaires à celle du Cloud Computing. L'objet de ces travaux était d'identifier les effets des attaques distribuées et de déterminer les limites des solutions traditionnelles de sécurité. Cette série d'expérimentations consistait à mener des attaques distribuées et à étudier la capacité des solutions de sécurité à détecter l'attaque en cours. Les résultats démontrent que les solutions traditionnelles ne sont pas efficaces face à des attaques modernes et que nous devons proposer de nouvelles techniques de détection d'intrusions, particulièrement dans des environnements où le trafic peut transiter par de nombreuses routes [RGH12a, RGH12b].

Ensuite, nos travaux se sont concentrés sur l'élaboration de Discus, un système distribué de sécurité, composé d'un grand nombre de sondes de sécurité. Notre contribution principale réside en la proposition d'un langage dédié, Discus Script, qui permet de décrire une politique de sécurité. Celle-ci sera ensuite analysée et optimisée, afin de pouvoir générer des fichiers spécifiques à chacune des sondes. Notre langage adopte une programmation événementielle et un typage fort. De plus, il propose une chaîne de compilation composée de nombreux outils, permettant la détection des incohérences, telle que la détection de cycles d'événements [RGH+14b, RGH14a].

Pour finir, nous nous sommes intéressés à l'élaboration d'un prototype du langage et des outils de génération de code, afin d'évaluer notre proposition. Pour cela, nous avons réalisé des campagnes d'études et de mesures, afin d'étudier l'expressivité du langage, sa robustesse. Nous avons également mesuré les performances de la solution générée, issue d'une politique écrite en Discus Script. Au travers de ces expérimentations, nous démontrons que notre langage est expressif et robuste, et que la solution générée est aussi efficace que des solutions existantes, voire meilleure en terme de consommation mémoire ou de temps de traitement. Cette vérification expérimentale permet de valider notre modèle de langage et de confirmer la pertinence de notre approche.

6.3 Perspectives

Les perspectives de ces travaux sont nombreuses et concernent l'ensemble des domaines étudiés. Nous listons dans les prochains paragraphes les perspectives que nous aurions souhaité aborder à la suite de cette thèse.

Élaboration d'un modèle distribué efficace La solution que nous proposons dans ce mémoire, Discus, présente deux axes principaux de travail : la configuration des sondes et la collaboration de celles-ci. Nos travaux s'intéressent principalement aux verrous scientifiques du premier thème, c'est pourquoi il serait intéressant d'étudier la distribution des sondes. Bien que nous mettons en place des mécanismes de collaboration, au travers du concept des tables, nos implémentations restent naïves et peu efficaces. L'étude des systèmes distribuées permettrait de mener des campagnes de simulations dans un environnement réel, impliquant un grand nombre de machines à surveiller. Cet axe de travail apporte son lot de problématiques, telles que la tolérance aux pannes, la détection de sondes byzantines ou la distribution équitable des traitements et données entre les éléments de la structure.

Support des circuits intégrés Il serait intéressant d'étudier d'autres types de sondes. Notamment, la cible initiale qui a motivé notre approche, le FPGA, serait une cible idéale pour ce domaine d'application. En effet, ces circuits intégrés proposent des performances très intéressantes et supportent naturellement la parallélisation de tâches. Toutefois, l'élaboration du générateur de la description du circuit intégré ne serait pas une tâche facile, car il serait tout

d'abord nécessaire d'acquérir une expertise de ce type de développement et d'adapter notre approche à cette nouvelle cible.

Extension du langage Il serait également intéressant d'étendre le langage pour qu'il s'adapte plus facilement à certaines tâches communes lors de la description d'une politique de sécurité. La syntaxe du langage devrait être étendue pour faciliter certaines actions, telles que la décomposition des protocoles en mode texte (comme HTTP, FTP ou SMTP) et la recomposition de flux TCP. À l'heure actuelle, la grammaire de Discus Script permet l'écriture de règles réalisant ces actions, mais cela ne nous semble pas assez naturel. Puisque notre volonté initiale est de proposer un langage simple à utiliser, il serait intéressant d'étudier les éléments de grammaire et de syntaxe qui faciliteraient ce type d'actions.

Reconfiguration à la volée des sondes Le domaine de la sécurité étant indubitablement en changement constant, il est nécessaire de s'intéresser aux mécanismes de reconfiguration à la volée, qui se concentre sur la mise à jour des données tout en conservant le système actif. Cette problématique est également applicable à la sécurité du Cloud Computing, où l'utilisateur peut souscrire à différents niveaux de services de sécurité. Dans ce cadre, il est nécessaire de maîtriser l'ensemble des opérations de cette structure, tel que la migration à chaud ou le démarrage et l'interruption de services. Ces problématiques vont de pair avec la reconfiguration à la volée des sondes, puisqu'il faut être capable de basculer de politique de sécurité de manière transparente, sans la moindre interruption des systèmes de sécurité.

Publications personnelles

[RGH12a] Damien Riquet, Gilles Grimaud et Michael Hauspie, « Étude de l'impact des attaques distribuées et multi-chemins sur les solutions de sécurité réseaux », *MajecSTIC 2012, 9ème Conférence Internationale Jeunes Chercheurs*, 2012.

[RGH12b] Damien Riquet, Gilles Grimaud et Michael Hauspie, « Large-scale coordinated attacks : Impact on the cloud security », *The Second International Workshop on Mobile Commerce, Cloud Computing, Network and Communication Security 2012*, p. 558, juil. 2012.

[RGH14a] Damien Riquet, Gilles Grimaud et Michaël Hauspie, « Discus : A massively distributed ids architecture using a dsl-based configuration », *in Information Science, Electronics and Electrical Engineering (ISEEE), 2014 International Conference on*, vol. 2, p. 1193–1197, IEEE, 2014.

[RGH+14b] Damien Riquet, Gilles Grimaud, Michaël Hauspie *et coll.*, « Un langage pour la configuration de discus, une architecture distribuée de solutions de sécurité », *in ComPAS*, 2014.

[Riq12] D. Riquet, « Massively distributed intrusion detection systems », 2012. Poster session, The European Professional Society on Computer Science, EuroSys 2012.

Bibliographie

[1] Amazon. Amazon web services : Overview of security processes. Technical report, 2011. URL http://aws.amazon.com/security.

[2] Debra Anderson, Teresa F Lunt, Harold Javitz, Ann Tamaru, Alfonso Valdes, et al. *Detecting unusual program behavior using the statistical component of the Next-generation Intrusion Detection Expert System (NIDES)*. SRI International, Computer Science Laboratory, 1995.

[3] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, and Matei Zaharia. Above the clouds : A berkeley view of cloud computing. Technical report, 2009.

[4] S. Axelsson. Intrusion detection systems : A survey and taxonomy. Technical report, Technical report, 2000.

[5] Charles Babcock. 9 worst cloud security threats, 2014. URL http://www.informationweek.com/cloud/infrastructure-as-a-service/9-worst-cloud-security-threats/d/d-id/1114085.

[6] Jai Sundar Balasubramaniyan, Jose Omar Garcia-Fernandez, David Isacoff, Eugene Spafford, and Diego Zamboni. An architecture for intrusion detection using autonomous agents. In *Computer Security Applications Conference, 1998. Proceedings. 14th Annual*, pages 13–24. IEEE, 1998.

[7] Jay Beale. *Snort 2.1 Intrusion Detection, Second Edition*. Syngress Publishing, 2004. ISBN 1931836043.

[8] S.M. Bellovin and W.R. Cheswick. Network firewalls. *Communications Magazine, IEEE*, 32 (9) :50 –57, sept. 1994.

[9] Steven M Bellovin. Distributed firewalls. *Journal of Login*, 24(5) :37–39, 1999.

[10] M. Bishop. What is computer security ? *Security Privacy, IEEE*, 1(1) :67–69, Jan 2003. ISSN 1540-7993. doi : 10.1109/MSECP.2003.1176998.

[11] Bitweasil. Cryptohaze cloud cracking. Defcon 20, 2012.

[12] David Bryan and Michael Anderson. Cloud computing : A weapon of mass destruction ?, 2010.

[13] Laurent Burgy, Laurent Réveillère, L. Lawall, Julia, and G. Muller. Zebu : A Language-Based Approach for Network Protocol Message Processing. *IEEE Transactions on Software Engineering*, 37(4) :575–591, 2011. URL https://hal.archives-ouvertes.fr/hal-00814448.

[14] Yanpei Chen, Vern Paxson, and Randy H. Katz. What's new about cloud computing security ? Technical Report UCB/EECS-2010-5, EECS Department, University of California, Berkeley, Jan 2010.

[15] Steven Cheung, Rick Crawford, Mark Dilger, Jeremy Frank, Jim Hoagland, Karl Levitt, Jeff Rowe, Stuart Staniford-Chen, Raymond Yip, and Dan Zerkle. The design of grids : A graph-based intrusion detection system. Technical report, 1999.

[16] Steven Cheung, Ulf Lindqvist, and Martin W Fong. Modeling multistep cyber attacks for scenario recognition. In *DARPA information survivability conference and exposition, 2003. Proceedings*, volume 1, pages 284–292. IEEE, 2003.

[17] William R Claycomb and Alex Nicoll. Insider threats to cloud computing : Directions for new research challenges. In *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual*, pages 387–394. IEEE, 2012.

[18] Fernando J Corbató, Marjorie Merwin-Daggett, and Robert C Daley. An experimental time-sharing system. In *Proceedings of the May 1-3, 1962, spring joint computer conference*, pages 335–344. ACM, 1962.

[19] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows : Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, volume 2, pages 119–129. IEEE, 2000.

[20] F. Cuppens and A. Miege. Alert correlation in a cooperative intrusion detection framework. In *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*, pages 202–215, 2002. doi : 10.1109/SECPRI.2002.1004372.

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

[22] Frédéric Cuppens. Managing alerts in a multi-intrusion detection environment. In *acsac*, page 0022. IEEE, 2001.

[23] Frédéric Cuppens and Rodolphe Ortalo. Lambda : A language to model a database for detection of attacks. In *Recent advances in intrusion detection*, pages 197–216. Springer, 2000.

[24] Oliver Dain and Robert K Cunningham. Fusing a heterogeneous alert stream into scenarios. In *Proceedings of the 2001 ACM workshop on Data Mining for Security Applications*, volume 13. Citeseer, 2001.

[25] H. Debar, M. Dacier, and A. Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 1999.

[26] Hervé Debar and Andreas Wespi. Aggregation and correlation of intrusion-detection alerts. In *Recent Advances in Intrusion Detection*, pages 85–103. Springer, 2001.

[27] Steve T. Eckmann, Giovanni Vigna, and Richard A Kemmerer. Statl : An attack language for state-based intrusion detection. *Journal of computer security*, 10(1/2) :71–104, 2002.

[28] European Network and Information Security Agency. Cloud computing risk assessment. Technical report, 2009.

[29] Maryam Feily, Alireza Shahrestani, and Sureswaran Ramadass. A survey of botnet and botnet detection. In *Emerging Security Information, Systems and Technologies, 2009. SECURWARE'09. Third International Conference on*, pages 268–273. IEEE, 2009.

[30] Jason Franklin and Vern Paxson. An inquiry into the nature and causes of the wealth of internet miscreants. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 375–388, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-703-2. doi : 10.1145/1315245.1315292. URL http://doi.acm.org/10.1145/1315245.1315292.

[31] Ayalvadi J Ganesh, Anne-Marie Kermarrec, and Laurent Massoulié. Peer-to-peer membership management for gossip-based protocols. *Computers, IEEE Transactions on*, 52(2) : 139–149, 2003.

[32] Joaquin Garcia, Fabien Autrel, Joan Borrell, Sergio Castillo, Frederic Cuppens, and Guillermo Navarro. Decentralized publish-subscribe system to prevent coordinated attacks via alert correlation. In *Information and Communications Security*, pages 223–235. Springer, 2004.

[33] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra : A virtual machine-based platform for trusted computing. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 193–206. ACM, 2003.

[34] Tal Garfinkel, Mendel Rosenblum, et al. A virtual machine introspection based architecture for intrusion detection. In *NDSS*, volume 3, pages 191–206, 2003.

[35] AliA. Ghorbani, Wei Lu, and Mahbod Tavallaee. Network attacks. In *Network Intrusion Detection and Prevention*, volume 47 of *Advances in Information Security*, pages 1–25. Springer US, 2010. ISBN 978-0-387-88770-8. doi : 10.1007/978-0-387-88771-5_1. URL http://dx.doi.org/10.1007/978-0-387-88771-5_1.

[36] Anna Giannakou, Louis Rilling, Jean-Louis Pazat, Frédéric Majorczyk, and Christine Morin. Towards Self Adaptable Security Monitoring in IaaS Clouds. In *ccgrid2015*, shenzen, China, May 2015. URL https://hal.inria.fr/hal-01165134.

[37] Ron Gula. Correlating ids alerts with vulnerability information, 2002.

[38] Simon Hansman and Ray Hunt. A taxonomy of network and computer attacks. *Computers and Security*, 24(1) :31 – 43, 2005. ISSN 0167-4048. doi : http://dx.doi.org/10.1016/j.cose.2004.06.011. URL http://www.sciencedirect.com/science/article/pii/S0167404804001804.

[39] L Todd Heberlein, Gihan V Dias, Karl N Levitt, Biswanath Mukherjee, Jefl Wood, and David Wolber. A network security monitor. In *Research in Security and Privacy, 1990. Proceedings., 1990 IEEE Computer Society Symposium on*, pages 296–304. IEEE, 1990.

[40] Cormac Herley and Dinei Florencio. Economics and the underground economy. Defcon 17, 2009.

[41] George V. Hulme. Amazon web services ddos attack and the cloud, 2009. URL http://www.darkreading.com/risk-management/amazon-web-services-ddos-attack-and-the-cloud/d/d-id/1083745?

[42] Kenneth Ingham and Stephanie Forrest. A history and survey of network firewalls. *University of New Mexico, Tech. Rep*, 2002.

[43] Ponemon Institute. Security of cloud computing providers study. Technical report, Ponemon Institute, 2011.

[44] Sotiris Ioannidis, Angelos D. Keromytis, Steve M. Bellovin, and Jonathan M. Smith. Implementing a distributed firewall. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, CCS '00, pages 190–199, New York, NY, USA, 2000. ACM. ISBN 1-58113-203-4. doi : 10.1145/352600.353052. URL http://doi.acm.org/10.1145/352600.353052.

[45] Kathleen A Jackson, David H DuBois, and Cathy A Stallings. An expert system application for network intrusion detection. Technical report, Los Alamos National Lab., NM (United States), 1991.

[46] V Jacobson, C Leres, and S McCanne. Libpcap initial release, 1994. URL http://www.tcpdump.org.

[47] Ramaprabhu Janakiraman, Marcel Waldvogel, and Qi Zhang. Indra : A peer-to-peer approach to network intrusion detection and prevention. In *Enabling Technologies : Infrastructure for Collaborative Enterprises, 2003. WET ICE 2003. Proceedings. Twelfth IEEE International Workshops on*, pages 226–231. IEEE, 2003.

[48] Jaeyeon Jung, Vern Paxson, Arthur W Berger, and Hari Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*, pages 211–225. IEEE, 2004.

[49] Min Gyung Kang, Juan Caballero, and Dawn Song. Distributed evasive scan techniques and countermeasures. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 157–174. Springer, 2007.

[50] Ramana Rao Kompella, Sumeet Singh, and George Varghese. On scalable attack detection in the network. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 187–200. ACM, 2004.

[51] Christopher Kruegel and William K Robertson. Alert verification determining the success of intrusion attempts. In *DIMVA*, volume 4, pages 1–14, 2004.

[52] Benjamin A Kuperman, Carla E Brodley, Hilmi Ozdoganoglu, TN Vijaykumar, and Ankit Jalote. Detection and prevention of stack buffer overflow attacks. *Communications of the ACM*, 48(11) :50–56, 2005.

[53] Michael E Locasto, Janak J Parekh, Sal Stolfo, and Vishal Misra. Collaborative distributed intrusion detection. 2004.

[54] Teresa F Lunt, R Jagannathan, Rosanna Lee, Sherry Listgarten, David L Edwards, Peter G Neumann, Harold S Javitz, and Al Valdes. Ides : The enhanced prototype-a real-time intrusion-detection expert system. In *SRI International, 333 Ravenswood Avenue, Menlo Park*. Citeseer, 1988.

[55] Peter Mell and Tim Grance. The NIST Definition of Cloud Computing. Technical report, July 2009.

[56] Julien Mercadal, Laurent Réveillère, Yérom-David Bromberg, Bertrand Le Gal, Tegawendé F. Bissyandé, and Jigar Solanki. Zebra : Building Efficient Network Message Parsers for Embedded Systems. *Embedded Systems Letters*, PP(99) :1–4, July 2012. doi : 10.1109/LES.2012.2208617. URL https://hal.archives-ouvertes.fr/hal-00730930. 4 pages.

[57] M. Mernik, J. Heering, and A.M. Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4) :316–344, 2005.

[58] Haroon Merr, Marco Slaviero, and Nicholas Arvanitis. Clobbering the cloud ! Defcon 17, 2009.

[59] Trend Micro. Public cloud fuels new breed of ddos attack, 2014. URL http://blog.trendmicro.com/public-cloud-fuels-new-breed-of-ddos-attack/.

[60] Chirag Modi, Dhiren Patel, Bhavesh Borisaniya, Hiren Patel, Avi Patel, and Muttukrishnan Rajarajan. A survey of intrusion detection techniques in cloud. *Journal of Network and Computer Applications*, 36(1) :42 – 57, 2013. ISSN 1084-8045. doi : http://dx.doi.org/10.1016/j.jnca.2012.05.003. URL http://www.sciencedirect.com/science/article/pii/S1084804512001178.

[61] Biswanath Mukherjee, L Todd Heberlein, and Karl N Levitt. Network intrusion detection. *Network, IEEE*, 8(3) :26–41, 1994.

[62] Gilles Muller, Julia L Lawall, Scott Thibault, and Rasmus Erik Voel Jensen. A domain-specific language approach to programmable networks. *Systems, Man, and Cybernetics, Part C : Applications and Reviews, IEEE Transactions on*, 33(3) :370–381, 2003.

[63] Ruoming Pang, Vern Paxson, Robin Sommer, and Larry Peterson. binpac : a yacc for writing application protocol parsers. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, IMC '06, pages 289–300, New York, NY, USA, 2006. ACM.

[64] Vern Paxson. Bro : a system for detecting network intruders in real-time. *Computer Networks*, 1999.

[65] Tao Peng, Christopher Leckie, and Kotagiri Ramamohanarao. Survey of network-based defense mechanisms countering the dos and ddos problems. *ACM Comput. Surv.*, 39(1), April 2007. ISSN 0360-0300. doi : 10.1145/1216370.1216373. URL http://doi.acm.org/10.1145/1216370.1216373.

[66] Phillip A Porras and Peter G Neumann. Emerald : Event monitoring enabling response to anomalous live disturbances. In *Proceedings of the 20th national information systems security conference*, pages 353–365, 1997.

[67] Phillip A Porras, Martin W Fong, and Alfonso Valdes. A mission-impact-based approach to infosec alarm correlation. In *Recent Advances in Intrusion Detection*, pages 95–114. Springer, 2002.

[68] John Postel. Transmission control protocol. RFC 793, Internet Engineering Task Force, September 1981. URL http://www.rfc-editor.org/rfc/rfc793.txt.

[69] OpenNebula Project. Opennebula, 2008. URL http://opennebula.org/.

[70] OpenStack Project. Openstack, 2010. URL https://www.openstack.org/.

[71] European Commission Expert Group Report. The future of cloud computing, 2010. URL http://cordis.europa.eu/fp7/ict/ssai/docs/cloud-report-final.pdf.

[72] Laurent Réveillère, Fabrice Mérillon, Charles Consel, Renuaud Marlet, and Gilles Muller. A dsl approach to improve productivity and safety in device drivers development. In *Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on*, pages 101–109. IEEE, 2000.

[73] Damien Riquet, Gilles Grimaud, and Michael Hauspie. Large-scale coordinated attacks : Impact on the cloud security. *The Second International Workshop on Mobile Commerce, Cloud Computing, Network and Communication Security 2012*, page 558, July 2012.

[74] Damien Riquet, Gilles Grimaud, and Michael Hauspie. Étude de l'impact des attaques distribuées et multi-chemins sur les solutions de sécurité réseaux. *MajecSTIC 2012, 9ème Conférence Internationale Jeunes Chercheurs*, 2012.

[75] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud : exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 199–212, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-894-0. doi : 10.1145/1653662.1653687. URL http://doi.acm.org/10.1145/1653662.1653687.

[76] M. Roesch et al. Snort-lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX conference on System administration*, pages 229–238. Seattle, Washington, 1999.

[77] Antony Rowstron and Peter Druschel. Pastry : Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001*, pages 329–350. Springer, 2001.

[78] P. Salgueiro and S. Abreu. Modeling distributed network attacks with constraints. *Intelligent Distributed Computing V*, pages 203–212, 2012.

[79] P. Salgueiro, D. Diaz, I. Brito, and S. Abreu. Using constraints for intrusion detection : the NeMODe system. *Practical Aspects of Declarative Languages*, 2011.

[80] P.D. Salgueiro. *Declarative Domain-Specific Languages and Applications to network monitoring*. PhD thesis, University of Evora, 2012.

[81] P.D. Salgueiro and S.P. Abreu. A dsl for intrusion detection based on constraint programming. In *Proceedings of the 3rd international conference on Security of information and networks*, pages 224–232. ACM, 2010.

[82] Stephen Shankland. Hp's hurd dings cloud computing, ibm, 2009. URL http://news.cnet.com/8301-30685_3-10378781-264.html.

[83] Vasilios A Siris and Fotini Papagalou. Application of anomaly detection algorithms for detecting syn flooding attacks. *Computer communications*, 29(9) :1433–1442, 2006.

[84] Stephen E Smaha. Haystack : An intrusion detection system. In *Aerospace Computer Security Applications Conference, 1988., Fourth*, pages 37–44. IEEE, 1988.

[85] Steven R. Snapp, , James Brentano, and Gihan V. Dias. Dids (distributed intrusion detection system) - motivation, architecture, and an early prototype. *in proceedings of the 14th National Computer Security Conference*, pages 167–176, 1991. doi : 10.1.1.46.4991.

[86] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing analysis of keystrokes and timing attacks on ssh. In *USENIX Security Symposium*, volume 2001, 2001.

[87] Eugene H Spafford. The internet worm program : An analysis. *ACM SIGCOMM Computer Communication Review*, 19(1) :17–57, 1989.

[88] Eugene H Spafford and Diego Zamboni. Intrusion detection using autonomous agents. *Computer networks*, 34(4) :547–570, 2000.

[89] Spiceworks. New study sees rise in cloud services adoption among small and medium businesses in first half of 2010, 2010. URL http://www.spiceworks.com/news/press-release/2010/07-28.php.

[90] Stuart Staniford, James A. Hoagland, and Joseph M. McAlerney. Practical automated detection of stealthy portscans. *J. Comput. Secur.*, 10(1-2) :105–136, July 2002. ISSN 0926-227X. URL http://dl.acm.org/citation.cfm?id=597917.597922.

[91] Stuart Staniford, David Moore, Vern Paxson, and Nicholas Weaver. The top speed of flash worms. In *Proceedings of the 2004 ACM Workshop on Rapid Malcode*, WORM '04, pages 33–42, New York, NY, USA, 2004. ACM. ISBN 1-58113-970-5. doi : 10.1145/1029618.1029624. URL http://doi.acm.org/10.1145/1029618.1029624.

[92] S. Staniford-chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagl, K. Levitt, C. Wee, R. Yip, and D. Zerkle. Grids - a graph based intrusion detection system for large networks. In *In Proceedings of the 19th National Information Systems Security Conference*, pages 361–370, 1996.

[93] tcpreplay. Tcpreplay - pcap editing and replaying utilities, 2015. URL http://tcpreplay.appneta.com/.

[94] Alfonso Valdes and Keith Skinner. Probabilistic alert correlation. In *Recent advances in intrusion detection*, pages 54–68. Springer, 2001.

[95] Fredrik Valeur. *Real-time intrusion detection alert correlation.* PhD thesis, UNIVERSITY OF CALIFORNIA Santa Barbara, 2006.

[96] Emmanouil Vasilomanolakis, Shankar Karuppayah, Max Mühlhäuser, and Mathias Fischer. Taxonomy and survey of collaborative intrusion detection. *ACM Comput. Surv.*, 47(4) :55 :1–55 :33, May 2015. ISSN 0360-0300. doi : 10.1145/2716260. URL http://doi.acm.org/10.1145/2716260.

[97] Giovanni Vigna and Richard A. Kemmerer. Netstat : A network-based intrusion detection system. *JOURNAL OF COMPUTER SECURITY*, 7 :37–71, 1999.

[98] Vivek Vishnumurthy and Paul Francis. On heterogeneous overlay construction and random node selection in unstructured p2p networks. In *INFOCOM*, pages 1–12. Citeseer, 2006.

[99] Haining Wang, Danlu Zhang, and Kang G Shin. Detecting syn flooding attacks. In *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1530–1539. IEEE, 2002.

[100] Nicholas Weaver, Vern Paxson, Stuart Staniford, and Robert Cunningham. A taxonomy of computer worms. In *Proceedings of the 2003 ACM Workshop on Rapid Malcode*, WORM '03, pages 11–18, New York, NY, USA, 2003. ACM. ISBN 1-58113-785-0. doi : 10.1145/948187.948190. URL http://doi.acm.org/10.1145/948187.948190.

[101] V. Yegneswaran, P. Barford, and S. Jha. Global intrusion detection in the domino overlay system. In *In Proceedings of Network and Distributed System Security Symposium (NDSS*, 2004.

[102] Saman Taghavi Zargar, Jyoti Joshi, and David Tipper. A survey of defense mechanisms against distributed denial of service (ddos) flooding attacks. *Communications Surveys & Tutorials, IEEE*, 15(4) :2046–2069, 2013.

[103] Chenfeng Vincent Zhou, Shanika Karunasekera, and Christopher Leckie. A peer-to-peer collaborative intrusion detection system. In *Networks, 2005. Jointly held with the 2005 IEEE 7th Malaysia International Conference on Communication., 2005 13th IEEE International Conference on*, volume 1, pages 6–pp. IEEE, 2005.

[104] Chenfeng Vincent Zhou, Shanika Karunasekera, and Christopher Leckie. Evaluation of a decentralized architecture for large scale collaborative intrusion detection. In *Integrated Network Management, 2007. IM'07. 10th IFIP/IEEE International Symposium on*, pages 80–89. IEEE, 2007.

[105] Chenfeng Vincent Zhou, Christopher Leckie, and Shanika Karunasekera. A survey of coordinated attacks and collaborative intrusion detection. *Computers and Security*, 29(1) :124 – 140, 2010.

[106] Minqi Zhou, Rong Zhang, Wei Xie, Weining Qian, and Aoying Zhou. Security and privacy in cloud computing : A survey. In *Semantics Knowledge and Grid (SKG), 2010 Sixth International Conference on*, pages 105–112. IEEE, 2010.

Table des figures

Liste des tableaux

Listings

A

Annexe 1 : Implémentation de protocoles en Discus Script

Dans cette annexe, nous présentons l'implémentation de plusieurs protocoles en Discus Script, tels que Ethernet, IPv4 ou TCP. Pour que l'exemple reste lisible, l'implémentation proposée ne contient que les éléments clés des protocoles.

Listing A.1– Implémentation basique du protocole Ethernet

```
1  # ------------------------
2  # ETHERNET RULES
3  # ------------------------
4
5  on packet(bitstream payload)
6      raise ethernet_packet(payload[48:95], payload[0:47], payload[96:111], payload[112:]);
7
8  on ethernet_packet(int48 mac_src, int48 mac_dst, int16 ethertype, bitstream payload)
9      where ethertype == 0x0800
10     raise ipv4_packet_raw(payload);
```

Listing A.2– Implémentation basique du protocole IPv4

```
# ------------------------
# IPV4 RULES
# ------------------------

on ipv4_packet_raw(bitstream payload)
    raise ipv4_packet(
        net4(payload[96:127], 32), # IPv4 src
        net4(payload[128:159], 32), # IPv4 dst
        payload[72:79], # Protocol
        payload[160:] # Payload
    );

on ipv4_packet(ipaddr ipv4_src, ipaddr ipv4_dst, int8 protocol, bitstream payload)
    where protocol == 0x06
    raise tcp_packet_raw(ipv4_src, ipv4_dst, payload);
```

Listing A.3– Implémentation basique du protocole TCP

```
# ------------------------
# TCP RULES
# ------------------------

on tcp_packet_raw(ipaddr ipv4_src, ipaddr ipv4_dst, bitstream payload)
    let data_offset = payload[96:99]
    raise tcp_packet(ipv4_src, ipv4_dst,
        payload[0:15], # Source port
        payload[16:31], # Dest port
        payload[32:63], # sequence number
        payload[64:95], # ack number
        payload[96:99], # data offset
        payload[103:111], # flags
        payload[112:127], # window size
        payload[128:143], # checksum
        payload[144:159], # urg pointer
        payload[32 * data_offset:]
    );


# ---- Enums
enum threshold_state {THRESHOLD_IN_PROGRESS, THRESHOLD_REACHED};
enum tcp_connection_state {TCP_HS_1, TCP_HS_2, ESTABLISHED, CLOSED};
enum tcp_flags {FIN=1, SYN=2, RST=4, PSH=8, ACK=16, URG=32, ECE=64, CWR=128, NS=256};

# ---- Table
table tcp_connection_table {
```

```
28      ipaddr client;
29      ipaddr server;
30      int16 p_client;
31      int16 p_server;
32      enum tcp_connection_state con_state;
33      time last_packet;
34  };
35
36  # -- Purge rule
37  on purge tcp_connection_table
38      select entry where ((now - entry.last_packet) > 60);
39
40  # -- Remove rule
41  remove entry from tcp_connection_table
42      when entry.con_state == tcp_connection_state.CLOSED;
43
44
45  # TCP Handshake: 1/3 (client -> (SYN) -> server)
46  on tcp_packet(ipaddr src, ipaddr dst, int16 src_port, int16 dst_port, int32
47      seq_num, int32 ack_num, int4 data_offset, int9 flags, int16 win_size, int16
48      checksum, int16 urg_ptr, bitstream payload)
49      where flags == tcp_flags.SYN
50      insert into tcp_connection_table {
51          client = src;
52          server = dst;
53          p_client = src_port;
54          p_server = dst_port;
55          con_state = tcp_connection_state.TCP_HS_1;
56          last_packet = now;
57      }
58      alert("tcp", "handshake 1/3");
59
60  # TCP Handshake: 2/3 (server -> (SYN|ACK) -> client)
61  on tcp_packet(ipaddr src, ipaddr dst, int16 src_port, int16 dst_port, int32
62      seq_num, int32 ack_num, int4 data_offset, int9 flags, int16 win_size, int16
63      checksum, int16 urg_ptr, bitstream payload)
64      where flags == (tcp_flags.SYN + tcp_flags.ACK)
65      for first t in tcp_connection_table with
66          (t.server == src) and
67          (t.client == dst) and
68          (t.p_server == src_port) and
69          (t.p_client == dst_port) and
70          (t.con_state == tcp_connection_state.TCP_HS_1)
71      update t.con_state = tcp_connection_state.TCP_HS_2
72      update t.last_packet = now
73      alert("tcp", "handshake 2/3");
74
```

```
75  # TCP Handshake: 3/3 (client -> (ACK) -> server)
76  on tcp_packet(ipaddr src, ipaddr dst, int16 src_port, int16 dst_port, int32
77      seq_num, int32 ack_num, int4 data_offset, int9 flags, int16 win_size, int16
78      checksum, int16 urg_ptr, bitstream payload)
79      where flags == tcp_flags.ACK
80      for first t in tcp_connection_table with
81          (t.client == src) and
82          (t.server == dst) and
83          (t.p_client == src_port) and
84          (t.p_server == dst_port) and
85          (t.con_state == tcp_connection_state.TCP_HS_2)
86      update t.con_state = tcp_connection_state.ESTABLISHED
87      update t.last_packet = now
88      alert("tcp", "handshake 3/3");
89
90  # TCP Closing connections: (client -> (FIN) -> server)
91  on tcp_packet(ipaddr src, ipaddr dst, int16 src_port, int16 dst_port, int32
92      seq_num, int32 ack_num, int4 data_offset, int9 flags, int16 win_size, int16
93      checksum, int16 urg_ptr, bitstream payload)
94      where flags == tcp_flags.FIN
95      for first t in tcp_connection_table with
96          (t.client == src) and
97          (t.server == dst) and
98          (t.p_client == src_port) and
99          (t.p_server == dst_port) and
100         (t.con_state == tcp_connection_state.ESTABLISHED)
101     update t.con_state = tcp_connection_state.CLOSED
102     update t.last_packet = now;
```