Orchestration et vérification de fonctions de sécurité pour des environnements intelligents

Nicolas Schnepf

UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Orchestration et vérification de fonctions de sécurité pour des environnements intelligents

THÈSE

présentée et soutenue publiquement le 30 septembre 2019

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Nicolas Schnepf

Composition du jury

Rapporteurs :	Christine Choppy, Professeur, LIPN, Université Paris 13, France Stefano Secci, Professeur, CNAM, France
Examinateurs :	Sandrine Vaton, Professeur, IMT Atlantique, France François Charoy, Professeur, Score - Loria, France Stephan Merz, Directeur de recherche, Veridis - Loria, France Remi Badonnel, Maître de conférences, Resist - Loria, France
Invité :	Abdelkader Lahmadi, Maître de conférences, Resist - Loria, France

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thesul.

Remerciements

En premier lieu, je tiens à remercier mes maîtres de thèse pour leur patience et leurs conseils tout au long des travaux de cette thèse. Je tiens également à remercier les rapporteurs pour leur travail de relecture de ce manuscrit. Je tiens aussi à remercier les instances de l'école doctorale IAEM pour leur accompagnement dans toutes les démarches administratives relatives au bon déroulement de la thèse. Je tiens enfin à remercier Pascal Fontaine pour sa figure décrivant l'architecture d'un SMT solver illustrant cette méthode dans le chapitre relatif à l'état de l'art.

Table des matières

Table des figures

1

Introduction

Ce chapitre a pour vocation d'introduire le questionnement propre à cette thèse portant sur l'orchestration et la vérification de fonctions de sécurité pour des environnements intelligents. Nous commençons par présenter le contexte de notre travail, à savoir la protection des environnements intelligents. Nous définissons également le concept de chaînes de fonctions de sécurité, et son usage dans ce cadre. Nous présentons ensuite les différents défis liés à l'orchestration de fonctions de sécurité, avant de décrire la problématique spécifique de la présente thèse et les trois axes sur lesquels elle s'articule. Finalement, nous terminons ce chapitre par l'annonce du plan du reste de ce manuscrit, qui suit la structure induite par les questionnements introduits dans ce chapitre.

1.1 Contexte de notre travail de recherche

Notre travail porte sur la protection d'environnements intelligents, et plus particulièrement les smartphones et tablettes. De manière générale, ces environnements peuvent être définis comme des équipements pouvant offrir des fonctionnalités proches de celles de postes fixes, tout en n'en ayant pas les mêmes ressources en termes de batterie ou de puissance de calcul. La protection de ces équipements est de ce fait difficile à implanter. Le recours à des méthodes de chaînage de fonctions de sécurité, qui peuvent être externalisées dans le cloud, offre des perspectives intéressantes à cet égard.

1.1.1 Sécurité des environnements intelligents

La protection des environnements intelligents est un enjeu majeur pour la cyber-sécurité. Comme indiqué précédemment, nous nous intéressons plus spécifiquement aux équipements mobiles à ressources limitées, que sont les smartphones et tablettes. En particulier, Android est devenu le premier système d'exploitation pour ces équipements avec 85,1 % de part du marché des smartphones en 2018 [58]. Le nombre d'applications disponibles sur les marchés correspondants, par exemple le Google Play Store, est marqué par une croissance exponentielle qui souligne la popularité de cette plateforme. Ceci introduit également des problèmes de sécurité du fait des applications malveillantes pouvant servir de vecteurs d'attaques [67]. En 2018, plus de 3 millions d'applications malveillantes ont été repéré sur les marchés dédiés au système d'exploitation Android [50], ce qui souligne bien l'enjeu en terme de cyber-sécurité.

Les menaces pesant sur les équipements intelligents sont de plusieurs natures. On rencontre, en premier lieu, les applications qui collectent les données personnelles des utilisateurs, avant de

les transmettre frauduleusement à des serveurs externes. Plusieurs cas de worms ont également été identifiés, il s'agit le cas échéant d'applications capables de s'auto-répliquer au travers d'un réseau en exploitant les vulnérabilités des systèmes d'exploitation. Ce mode d'infection des équipements intelligents peut par exemple être utilisé afin de les transformer en machines zombies en vue de les enrôler dans un botnet, un réseau de machines corrompues. De tels réseaux peuvent ensuite être utilisés afin de pratiquer des attaques de dénis de service, ou des scans de ports à large échelle. Finalement, on repère aussi de nombreux cas de rançonwares dans lesquels une application va chiffrer le contenu d'un équipement et proposer le déchiffrement moyennant une certaine somme de laquelle l'utilisateur devra s'acquitter.

Le développement de ces menaces peut s'expliquer par plusieurs facteurs. D'abord, il est important de souligner l'insuffisance des méthodes dites préventives actuellement déployées au niveau des marchés (e.g. Google Play Store). Ces méthodes consistent à valider la sûreté et la sécurité d'une application à partir des interfaces logicielles qu'elle propose ainsi qu'au travers d'exécutions symboliques ou simulées en environnements contrôlés. Ces méthodes permettent ainsi de détecter des applications potentiellement malveillantes pour les utilisateurs. Néanmoins, au regard des chiffres cités plus haut, il ressort clairement que : si elles sont nécessaires, elles sont loin d'être suffisantes pour assurer la protection de ces environnements. La première limitation propre aux méthodes dites préventives provient du fait qu'elles procèdent à une validation automatisée et standardisée des applications, et qu'à ce titre certaines failles peuvent échapper à leur vigilance. En outre, les méthodes dites préventives permettent la détection d'applications constatées comme malveillantes, lors de leur mise à disposition sur les marchés. En revanche, dans le cas où la partie malveillante se trouve comme souvent être dissimulée dans une mise à jour, elles restent inopérantes. Enfin, dans le cas où l'utilisateur aurait installé une application provenant d'une autre source que les marchés dédiés ou dans le cas où plusieurs applications distinctes s'associeraient en vue de l'exécution d'une manoeuvre malveillante, les méthodes préventives s'avèrent démunies, d'où la nécessité de les compléter par des méthodes réactives qui assurent la protection des équipements intelligents pendant leur exécution.

Dans ce contexte, le déploiement de fonctions de sécurité telles que des pare-feux ou des antivirus, peut s'opérer directement sur lesdits équipements. Cette proposition découle de la supposition partiellement étayée dans les faits que les équipements intelligents sont des formes d'ordinateurs portatifs offrant les mêmes services que leurs homologues fixes. Il semble naturel de protéger ces deux types d'équipements de la même manière. Il est ainsi possible de télécharger la version mobile de nombreux services de sécurité, tels que les antivirus connus du marché, pour assurer la protection de son smartphone ou de sa tablette. Néanmoins, cette approche souffre de limitations propres de ces environnements. En effet, l'usage de méthodes traditionnelles est rendue difficile par les fortes contraintes en termes de batterie et de CPU de ces équipements. Le déploiement d'antivirus ou de pare-feux dédiés sur les smartphones et les tablettes risque premièrement de fortement dégrader l'expérience des utilisateurs, en limitant drastiquement le temps d'utilisation qu'ils pourront en attendre, ou en introduisant de fortes latences dans l'exécution des traitements attendus. En outre, le déploiement de fonctions de sécurité directement sur les équipements intelligents pose aussi la difficulté liée au fait qu'elle requiert le téléchargement de services auquel l'utilisateur doit se fier au risque qu'il s'agisse en fait d'applications malveillantes, comme décrit ci-dessus.

1.1.2 Chaînes de fonctions de sécurité

Les travaux relatifs au concept de chaînes de fonctions de sécurité s'inscrivent dans le cadre plus large de la recherche sur la programmabilité et la virtualisation de fonctions réseau [26].

Ces méthodes visent à proposer différentes fonctions réseau assurant des tâches de sécurité ou de gestion comme des services logiciels indépendamment de leur implantation physique. Cette approche vise à déléguer l'allocation de fonctions réseau à des orchestrateurs logiciels afin de rendre ces aspects de configuration transparents pour les utilisateurs du réseau. De telles fonctions réseau virtualisées sont couramment appelées VNF pour *Virtualized Network Functions*. Les middleboxes sont un équivalent hardware des VNFs, il s'agit d'équipements réseau qu'un opérateur peut composer pour assurer certains services, notamment en sécurité.

C'est dans ce contexte qu'a été proposée l'utilisation de chaînes de sécurité pour la protection des environnements intelligents [56, 57, 76]. Ces chaînes sont des compositions dynamiques de fonctions de sécurité, telles que des pare-feux, des systèmes de détection d'intrusion (IDS), ou des services de prévention de fuite de données [62, 77, 103]. Ces fonctions de sécurité sont virtualisées et exposées comme des middleboxes [38, 9] au sein d'un réseau. La construction de chaînes de fonctions de sécurité peut ainsi être vue comme un service assuré par les fournisseurs de réseau à leurs abonnés. Les fonctions de sécurité peuvent ainsi être composées dynamiquement afin de constituer une ou plusieurs chaînes de fonctions de sécurité pouvant être utilisées pour la protection de systèmes critiques. Ces chaînes de fonctions de sécurité peuvent être dynamiquement construites à la demande des utilisateurs et être proposées comme un service cloud.

Plus précisément, les tâches de construction, d'optimisation et de déploiement de chaînes de fonctions de sécurité peuvent être centralisées et automatisées par un orchestrateur dédié. Cet orchestrateur sera alors contacté par tous les utilisateurs nécessitant une chaîne de fonction de sécurité pour assurer leur protection. En fonction de l'état du réseau, l'orchestrateur pourra alors soit construire une nouvelle chaîne répondant aux besoins du nouvel utilisateur, soit mettre à jour une chaîne existante soit rediriger l'utilisateur vers une de ces chaînes. Pour assurer un tel service, un orchestrateur de chaînes de fonctions de sécurité doit être en mesure d'identifier les besoins particuliers d'un utilisateur, de construire la chaîne correspondante et d'en optimiser le placement avant de la déployer effectivement dans le réseau.

Toutefois, le principal inconvénient des chaînes de fonctions de sécurité réside dans leur grande complexité structurelle ainsi que dans leur dynamique qui risquent d'introduire des erreurs de configuration dans le réseau, par exemple des règles associant des actions contradictoire pour le même traffic. En effet, les différentes fonctions de sécurité utilisées dans le déploiement de chaînes de sécurité présentent des interfaces souvent fort différentes et l'harmonisation de leurs configurations n'est pas un problème trivial. Pour répondre à cela, plusieurs approches ont été proposées dont le recours à des méthodes formelles pour permettre la validation des chaînes avant leur déploiement : ces méthodes permettent en effet la validation de systèmes critiques en travaillant sur une représentation abstraite de leur comportement. Le principal inconvénient de ces méthodes réside dans leur complexité algorithmique qui requiert un travail attentif afin de permettre leur application dans un contexte d'orchestration.

1.2 Défis et problématique

L'utilisation des technologies associées aux chaînes de fonctions de sécurité soulève des problèmes multiples. La virtualisation a été proposée comme une alternative à l'utilisation des middleboxes, difficiles à gérer et souvent propriétaires. Toutefois, l'utilisation d'un composant logiciel en lieu et place d'un équipement matériel propre n'est pas sans introduire de nouveaux défis. Nous présentons ici les défis propres à l'orchestration de fonctions de sécurité dans le réseau. Nous décrivons ensuite notre approche pour la composition, la validation et l'optimisation automatiques de chaînes de telles fonctions en environnement Software Defined Network (SDN).

1.2.1 Défis liés à l'orchestration de chaînes de fonctions de sécurité

Un premier défi porte sur l'administration des fonctions virtualisées dans le contexte d'une architecture opérateur. En effet, les fournisseurs de réseau sont intéressés par la possibilité d'optimiser leur service à moindre coût de déploiement. Il est cependant à craindre que l'introduction des solutions de virtualisation, plus flexibles, introduisent aussi une plus grande complexité dans la gestion du réseau. Un intérêt grandissant autour des technologies clouds est celui de réseau en tant que service : l'idée est de pouvoir fournir des services à la demande aux utilisateurs. Dans un tel contexte, on peut légitimement se poser la question de la faisabilité d'une telle approche, fournir un service à la demande de chaque utilisateur pouvant sembler difficilement envisageable d'un point de vue logistique. S'il n'est effectivement pas possible de satisfaire individuellement toutes les demandes, alors peut-être serait-il possible de les factoriser afin de les traiter collectivement.

Un second défi touche à la résilience des technologies de virtualisation : en effet, comme nous l'avons vu plus haut le déploiement de multiples fonctions de sécurité configurées en lien les unes avec les autres risque d'introduire des erreurs dans le réseau. Les approches basées sur des tests unitaires ne sont pas suffisantes pour permettre la validation de systèmes complexes tels que les chaînes de fonctions virtualisées. En effet, leur structure essentiellement dynamique les rend difficile à valider par l'exécution d'une série de tests statiques, les erreurs n'ayant pas lieu à chaque exécution. Il est en outre complexe de valider l'absence de contradictions des règles d'une chaîne par application de tests unitaires, le résultat des tests pouvant correspondre au comportement attendu, alors que le programme de la chaîne comporte bel et bien une incohérence. Peut-être les méthodes formelles peuvent permettre de contourner ces difficultés, si tel est le cas alors il reste encore la problématique de leurs performances. En effet, garantir et optimiser une validation en temps réel, c'est-à-dire sans interrompre l'exécution du réseau, des chaînes de fonctions de sécurité sur la base de méthodes de vérification formelle est un enjeu important. La composition et le déploiement des chaînes de sécurité doivent être assurés à chaque fois qu'un équipement le demande. Il est donc nécessaire d'intégrer les méthodes de vérification formelle dans un cycle d'exploitation en temps réel, afin de assurer les besoins en termes de validation.

Un dernier défi important concerne les performances des technologies de virtualisation elles-mêmes. En effet, l'introduction de composants logiciels dans le réseau en lieu des composants matériels risque d'être à l'origine d'une plus grande complexité de gestion. Cette problématique influe directement sur les performances liées au déploiement de fonctions virtualisées : en effet, l'efficacité des réseaux dépend grandement de leur capacité à fournir de très hautes qualités de service en termes de bande passante et de temps de réponse. Dans ce contexte, on peut légitimement se poser la question de l'impact des technologies de virtualisation, non plus sur le déploiement, mais sur l'utilisateur final, l'utilisation de composants logiciels risque en effet d'apporter des délais dégradant la qualité du service, voire même d'introduire des failles de sécurité supplémentaires.

1.2.2 Problématique relative à cette thèse

Notre travail dans le cadre de cette thèse se focalise sur l'application et l'exploitation des méthodes formelles telles que le model checking, le SMT solving ou la programmation logique pour une orchestration plus sûre des fonctions de sécurité en environnement SDN. En particulier, les chaînes de sécurité ainsi construites peuvent être utilisées en vue d'assurer la protection d'équipements intelligents. Cet objectif peut être structuré autour de trois axes principaux : la conception d'un orchestrateur dédié à la construction des chaînes de fonctions de sécurité,

l'intégration de méthodes de vérification automatique dans cette architecture, pour permettre la validation des chaînes en temps réel, et enfin les méthodes qui peuvent être employées en vue d'optimiser le déploiement des chaînes de fonctions de sécurité dans le réseau.

Le premier axe porte sur la conception architecturale d'un orchestrateur en charge de la gestion des chaînes. Le but que nous avons assigné pour notre orchestrateur est de permettre le déploiement dynamique de chaînes de fonctions de sécurité pour protéger des smartphones et tablettes dont les ressources sont relativement restreintes. Dans ce contexte, notre orchestrateur se doit d'assurer des performances satisfaisantes en termes de sécurité et de temps de réponse, un souci tout particulier doit être attaché à cet aspect dans la conception de son architecture. Le déploiement de notre orchestrateur au sein du réseau, ainsi que ses interactions avec les diverses entités le constituant seront également discutées ultérieurement dans ce mémoire.

Le second axe est celui de l'intégration de méthodes formelles comme un support à notre orchestrateur. Comme nous l'avons déjà brièvement évoqué, les méthodes formelles offrent de grandes opportunités en vue de la validation de systèmes complexes, tels que les chaînes de fonctions de sécurité, mais peuvent également être d'une très forte complexité algorithmique, qui en restreint souvent l'application en temps réel. La caractérisation des chaînes de fonctions de sécurité et des diverses propriétés devant être validées au travers des méthodes formelles seront abordées ultérieurement dans ce mémoire. Nous nous intéresserons notamment à l'identification des méthodes offrant les meilleures performances en vue d'assurer la validation des chaînes de sécurité en temps réel.

Enfin, le dernier axe de notre travail tient aux possibilités en termes d'optimisation des chaînes avant leur déploiement. La capacité des réseaux n'étant pas illimitée, il est en effet essentiel d'assurer que le déploiement des chaînes de sécurité n'induit pas de surcoûts excessifs au niveau des opérations devant être réalisées. Deux aspects principaux seront examinés, premièrement la minimisation du nombre de fonctions de sécurité à déployer, avec éventuellement leur factorisation au sein d'un ensemble de chaînes. Le second aspect traitera de l'optimisation de leur placement en fonction de différents paramètres contextuels induits par le réseau.

1.3 Plan de la thèse

La présente thèse s'organise en fonction des axes de recherche décrits dans la section précédente. L'axe relatif à la conception d'un orchestrateur en charge de la gestion des chaînes de fonctions de sécurité est abordé dans le chapitre 3 qui traite en particulier de l'apprentissage du comportement des applications en vue de construire les chaînes correspondantes. L'axe relatif à l'exploitation de méthodes formelles pour la validation à l'exécution desdites chaînes de fonctions de sécurité est traité dans les chapitres 4 et 5. Finalement, l'axe relatif à l'optimisation du déploiement des chaînes de fonctions de sécurité est abordé dans le chapitre 6.

Le chapitre 2 fait tout d'abord un état de l'art des travaux liés à la présente thèse. En particulier, il décrit les travaux concernant la sécurité des réseaux, et plus particulièrement celle des équipements intelligents. Il présente ensuite les travaux relatifs aux technologies de réseaux programmables (SDN), et à celles de virtualisation et de middleboxes. Le chapitre décrit également les travaux existants en matière de méthodes formelles, et plus spécifiquement à leur application à la validation de chaînes de fonctions de sécurité. Nous abordons également les différents travaux traitant de l'apprentissage automatique de processus et de protocoles réseau, ainsi que des méthodes permettant la résolution de problèmes d'optimisation.

Le chapitre 3 présente l'architecture générale de notre orchestrateur et son intégration au sein d'un réseau SDN. Il détaille les différents modules impliqués dans la génération et la gestion

des chaînes, et offre ainsi un fil conducteur qui sera utile au lecteur pour comprendre la logique des chapitres suivants, placés dans l'ordre des traitements de notre orchestrateur. Ce chapitre décrit en particulier l'apprentissage du comportement réseau d'applications Android. En particulier, il présente le type d'automates markoviens que nous avons retenu pour représenter le comportement des applications, ainsi que les informations à partir desquelles ces modèles sont construits. Le chapitre se termine par le comparatif de notre approche avec d'autres méthodes analogues et présente les avantages induits par notre processus d'apprentissage dans le contexte d'application Android décrites par des flux réseau.

Le chapitre 4 détaille notre approche de synthèse formelle de chaînes de fonctions de sécurité. Elle s'appuie sur les automates markoviens décrits au chapitre 3, à partir desquels nous construisons une spécification logique des besoins de sécurité. Cette spécification est ensuite utilisée par un système d'inférence basé sur des clauses de Horn pour générer la chaîne de fonctions de sécurité dédiée à une application Android. Nous faisons la preuve de plusieurs propriétés ainsi garanties par lesdites chaînes. Le chapitre 4 présente également comment de telles chaînes peuvent être factorisées afin de en minimiser le coût de déploiement. Finalement, il se termine en décrivant les résultats expérimentaux que nous avons obtenus en générant les chaînes associées à un ensemble d'applications données, et en comparant différentes méthodes de factorisation.

Le chapitre 5 aborde notre approche de vérification automatique de chaînes, couvrant à la fois la vérification du plan de contrôle et du plan de données. Il présente les composants que nous proposons pour assurer une telle vérification, reposant notamment sur des méthodes de traduction automatique pour convertir la spécification haut niveau des chaînes de fonctions de sécurité, en modèles formels pouvant être vérifiés automatiquement par des solvers dédiés. Ce chapitre présente les différents types de modélisation que nous avons appréhendés, notamment les représentations basées sur des méthodes de model checking et de SMT solving, et compare les performances apportées par ces différentes méthodes pour la vérification automatique de chaînes de sécurité.

Le chapitre 6 indique ensuite le résultat de nos travaux touchant au placement de chaînes de fonctions de sécurité dans un contexte SDN. Ce chapitre commence par motiver le problème en détaillant la difficulté du placement des chaînes dans un réseau présentant certaines contraintes, puis introduit deux abstractions permettant la simplification du modèle. Il se poursuit en détaillant les divers éléments relatifs au modèle de placement, notamment ses paramètres et ses variables, ses contraintes et ses objectifs. Le chapitre inclut le comparatif des performances assurées par différents outils dédiés à l'optimisation formelle. Nous discutons finalement les résultats observés après avoir simulé le déploiement de nos chaînes de fonctions de sécurité avec le simulateur de réseau SDN MiniNet.

Enfin, le chapitre 7 est dédié aux conclusions de notre travail dans le cadre de cette thèse. Il commence par résumer les contributions, puis apporte une réponse aux questionnements liées à la problématique. Le chapitre se termine sur la présentation de différentes perspectives de recherche. En particulier, il pourrait être d'un vif intérêt d'étendre la spécification des règles que nous utilisons pour configurer les chaînes de fonctions de sécurité, ainsi que les méthodes de détection auxquelles nous avons eues recours.

La figure 1.1 présente les relations entre les différents chapitres de contributions décrits dans ce mémoire (hors chapitre 2 consacré à l'état de l'art). Le chapitre 3 constitue le point de départ de notre approche d'orchestration. Il est ensuite possible d'aborder le chapitre 4 dans le cas d'une approche où les chaînes sont générées automatiquement à partir des automates markoviens décrivant le comportement des applications, puis factorisées pour minimiser l'impact de leur déploiement sur le réseau. Il est également possible d'aborder le chapitre 5 dans lequel une chaîne peut être spécifiée manuellement par un opérateur réseau sans recours à des méthodes

de synthèse. Le chapitre 6 clôt notre démarche d'orchestration en apportant les éléments relatifs au placement des règles d'une chaîne avant son déploiement ainsi que l'évaluation du surcoût ainsi induit.

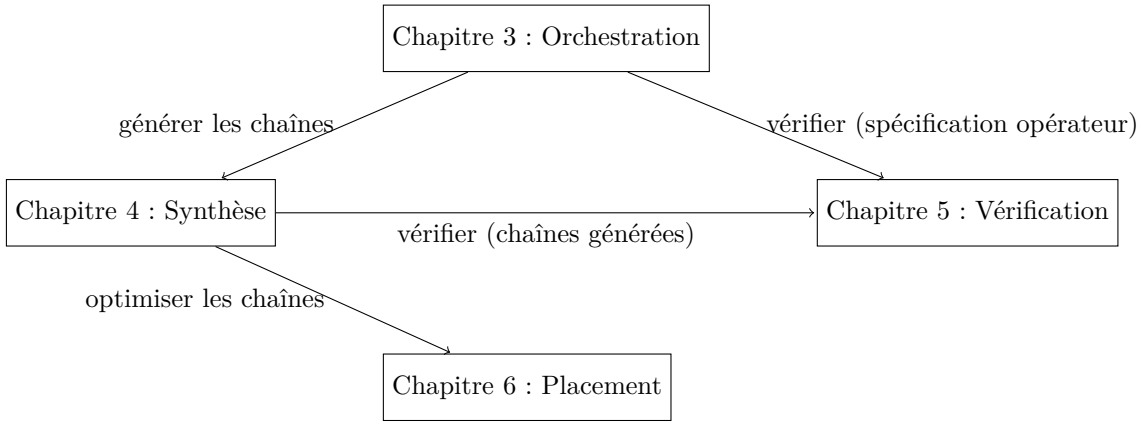

FIGURE 1.1 – Diagramme des relations entre les chapitres de contributions de la thèse

2

Etat de l'art

De nombreux travaux ont d'or et déjà été menés dans les différents domaines que nous explorons dans le cadre de ce mémoire. Ce chapitre présente les travaux relatifs au contexte de notre étude, en particulier les travaux portant sur la sécurité des réseaux, et plus particulièrement celle des environnements intelligents, les travaux relatifs aux solutions s'appuyant sur la programmabilité et virtualisation réseau, et enfin les travaux relatifs aux différentes méthodes formelles que nous avons utilisées au cours de la thèse.

2.1 Sécurité des équipements intelligents

Comme indiqué en introduction le but de notre travail est d'assurer la protection d'équipements intelligents, tels que des smartphones ou des tablettes, en particulier les équipements gérés par le système d'exploitation Android. Nous présentons ici les travaux relatifs aux menaces ciblant ces équipements, ainsi que ceux relatifs aux solutions qui ont été proposées pour y remédier. Notre étude s'articule en plusieurs volets suivant la classification définie dans [99], à savoir d'abord la sécurité au niveau du réseau, ensuite les travaux relatifs aux permissions du système Android, et finalement les techniques de profilage des applications en vue de leur sécurisation.

2.1.1 Menaces réseau contre les équipements intelligents

Notre travail se concentre plus particulièrement sur la protection des équipements intelligents au niveau du réseau. Dans [67] sont présentées plusieurs des difficultés liées au domaine, en particulier les hautes contraintes en termes de batterie et de processeur. Des problématiques plus étroitement liées à la protection d'équipements Android sont présentées dans [39], en particulier les problèmes liés à l'explosion d'applications malveillantes sur les marchés dédiés. Ces travaux ne proposent toutefois pas eux-mêmes de solutions, ils se limitent à faire un récapitulatif des travaux existants concernant les menaces visant ces environnements.

Les équipements intelligents sont la cible de plusieurs types d'attaques telles que par exemple des dénis de service, des scans de ports, des worms ou encore des botnets, comme indiqué dans [67]. Une attaque par *déni de service* (DoS, *Denial of Service*) est caractérisée par une ou un ensemble de machine(s) ciblant une victime pour l'empêcher d'assurer un service [52]. Dans notre contexte, nous considérons des attaques par DoS que l'on peut observer d'un point de vue réseau, c'est-à-dire produisant des modifications anormales du traffic en partance ou à destination d'un certain équipement. Un exemple d'une telle attaque est l'attaque SYN flood, dans laquelle un grand nombre de paquets SYN sont envoyés à un hôte, afin de déborder la pile TCP

avec des centaines de connexions qui ne seront jamais refermées. Dans une attaque par scan de ports (ou *port scanning*), une application initie un grand nombre de connexions avec les ports d'une ou de plusieurs machine(s) afin de détecter les ports ouverts. Nous considérons des scans de ports comparables à ceux générés par le scanner de ports nmap disponible sur les plateformes linux standards. Un ver (ou *worm*) est un programme qui peut s'exécuter indépendamment tout en consommant les ressources de son hôte afin de se maintenir, et peut répliquer une version complètement exécutable de lui-même vers d'autres machines [71]. Les worms se répliquent en exploitant les vulnérabilités d'applications et systèmes d'exploitation, ou par des méthodes d'ingénierie sociale. Dans cette thèse nous considérons des worms qui scannent un certain port sur de nombreuses machines, afin de exploiter une certaine vulnérabilité sur des systèmes d'exploitation. Un *bot* (potentiellement) malveillant est un programme installé sur un système afin de l'utiliser automatiquement ou semi-automatiquement en vue d'exécuter des tâches, potentiellement sous le contrôle d'un administrateur distant, appelé *bot master* [14]. La détection de tels botnets est un domaine de recherche à part entière du fait de leurs architectures complexes et de la diversité de leurs implémentations. Il existe en particulier des botnets communiquant sur la base de requêtes http difficilement identifiables d'un point de vue réseau ou encore des botnets exploitant une architecture pair à pair dans laquelle les bots se relayent les messages du bot master de proche en proche. Dans le cadre de notre travail nous nous limiterons au cas de bots pouvant être détectés sur la base du fort volume de trafic qu'ils échangent avec leur contrôleur ou potentiellement par l'utilisation de protocoles réseau inhabituels dans un certain contexte.

Ces différentes attaques peuvent être détectées au niveau des paquets réseaux qu'elles génèrent ou bien au niveau des flux comme indiqué dans [98]. D'après le RFC 5101 [27], les flux réseau peuvent être définis comme "des collections de paquets IP passant par un certain point d'observation dans le réseau pendant un certain intervalle de temps". Ils sont généralement décrits par différentes propriétés telles que les adresses IP source et destination (*srcaddr* et *dstaddr*), les ports source et destination (*srcport* et *dstport*), leur protocole réseau (*protocol*), et leur nombres de paquets et d'octets (*packets* et *bytes*). Dans notre contexte ils sont collectés directement sur les équipements [68], et sont étendus avec un timestamp (*timestamp*) et le nom de l'application qui les a produits (*appname*). La limitation de ces travaux relatifs aux flux réseau est qu'ils ne prennent pas en compte la modélisation des données transmises lors d'une communication et qu'ils doivent donc être complétés par des travaux appréhendant ces problématiques, par exemple via le système de permissions Android. Comme indiqué dans [98] la recherche sur les flux réseaux requiert la constitution de volumineux datasets contenant de nombreuses traces décrivant le comportement malveillant ou non d'hôtes et d'applications. De semblables datasets ont été proposés par des chercheurs du domaine public, par exemple le *CTU13* présenté dans [49] qui contient de nombreuses traces de botnets générées au sein d'un réseau universitaire. Néanmoins la constitution de nouveaux datasets contenant des traces de DoS, de scans de ports ou encore de worms reste un enjeu pour la recherche en ces domaines. Enfin il est également nécessaire de présenter les attaques par fuite de données qui constituent une autre menace sérieuse contre les équipements intelligents. Ces attaques visent à violer la confidentialité de certaines données privées d'un utilisateur en les transmettant sans son accord à un tiers malfaisant. Dans [83], les auteurs proposent une approche permettant la détection de fuites des données privées d'un utilisateur via l'observation des flux réseau générés par une application. Toutefois, dans les contextes où la majeure partie du trafic émis peut être chiffrée avec l'usage du protocole HTTPS, il devient difficile d'utiliser de telles méthodologies. Il convient donc d'avoir recours à d'autres moyens pour identifier de potentielles fuites de données.

2.1.2 Système de permissions des environnements Android

L'utilisation du système d'exploitation Android est régi par de nombreuses applications qu'un utilisateur peut installer via le Google Play Store, ou bien directement via leurs exécutables. Les droits accordés à ces applications sur les diverses ressources de l'équipement, telles que par exemple la connexion à Internet ou l'utilisation de l'appareil photo sont régis par un système de permissions dédiées. Les permissions requises par une application sont spécifiées dans un fichier de manifest incorporé à l'archive contenant l'exécutable. Cette archive peut également contenir d'autres fichiers de méta-données tels que des images utilisées par l'application. Comme indiqué dans [99], la protection d'applications Android demande l'examen des permissions demandées. Le système d'exploitation Android repose en effet sur un système de permissions qui permettent de restreindre l'accès à certaines ressources critiques en termes de sécurité tel que par exemple l'appareil photo ou la liste des contacts d'un utilisateur. Bien que détaillé par diverses documentations techniques [1], le système de permissions d'Android reste source de nombreuses erreurs du fait de son peu de lisibilité. Plusieurs travaux ont été proposés pour permettre une meilleure lisibilité des permissions régissant l'activité des applications Android. Dans [10], les auteurs proposent une approche faisant l'association entre les permissions déclarées par le système de sécurité d'Android et les méthodes déclarées dans l'API [1] correspondante. Cette approche permet la surveillance du comportement d'applications et en particulier la détection d'accès frauduleux aux données d'un utilisateur, toutefois il reste à déplorer que cette approche ne prenne pas en compte la protection au niveau réseau. En effet l'obtention de la permission internet par une application Android lui ouvre un plein accès à cette ressource et ne permet pas en soi d'identifier de potentiels comportements malveillants.

L'une des principales règles conditionnant le développement d'applications sûres est le respect du principe de moindre privilège : dans [81], les auteurs proposent une approche visant à identifier des applications sur-privilégiées. Cette approche propose de faire l'association entre les permissions déclarées par une application et les appels de méthode effectivement identifiés dans le binaire de ladite application. Ce travail a été évalué de manière préventive sur un large ensemble d'applications disponibles sur le Google play store. La conclusion des auteurs est que malgré les efforts mis en oeuvre environ un tiers des applications actuellement déployées au niveau des marchés bénéficient de trop grands privilèges. Toutefois ils ne proposent pas de solution remédiant à ce problème ou permettant aux utilisateurs ayant installé lesdites applications de se prémunir d'éventuels comportements malveillants. Plusieurs travaux couvrent également l'analyse du comportement d'applications Android en vue de détecter des comportements malveillants liés à la demande de certaines permissions. Dans [7], les auteurs proposent une approche visant à détecter la collusion d'applications en vue d'accomplir des actions malveillantes qui n'auraient pas été possibles si les permissions qu'elles déclarent n'avaient été déclarées que par l'une d'entre elles, par exemple extruder des données privées du carnet d'adresse et les transmettre à un serveur distant via une autre application. On peut néanmoins déplorer le fait que ce travail ne couvre pas tous les aspects de communications Android, par exemple la communication via des *content sets* dédiés. De même, les travaux présentés dans [51] tendent à permettre l'identification d'applications malveillantes dont le comportement dévie de celui attendu au vue des permissions et du comportement qu'elles déclarent. Cette approche se base sur le descriptif des applications fournis par le développeur, ainsi que sur le comportement réellement observé lors de l'exécution de ladite application en environnement protégé. Appliqué sur un ensemble de 22500 applications Android, le prototype détaillé dans cette approche identifia 56% de nouveaux virus sans disposer d'aucun modèle préalable en vue de cette détection.

1. Application Programming Interface

Les approches détaillées dans cette sous-section analysent le comportement des applications de manière préventive. Bien que ces approches préventives trouvent tout leur sens dans la protection d'applications mobiles, il reste néanmoins nécessaire de disposer de modèles de leur comportement à l'exécution afin de permettre une meilleure adaptation de la sécurité.

2.1.3 Profilage du comportement d'applications Android

L'établissement de profils d'applications Android a été exploré par de nombreux travaux. Dans [99], les auteurs proposent un outil de profilage travaillant à plusieurs couches du système Android. Cette approche se démarque des autres en ce qu'elle appréhende le comportement d'une application à 4 niveaux différents : la spécification du comportement de l'application, les appels de méthodes réalisés au niveau système, les interactions avec l'utilisateur, et enfin le trafic réseau généré par l'application. Cette approche a été évaluée avec un ensemble de 27 applications Android téléchargées depuis le Google Play Store, ou bien obtenues à partir de sources tierces. Ces travaux ont révélé que le comportement de nombreuses applications Android ne correspond pas à leur spécification, de plus il ressort que la majeure partie du trafic émis n'est pas chiffrée. Enfin, ce travail a révélé que de nombreuses applications contactent davantage de destinations que ce qui est déclaré dans leur spécification, en particulier de nombreuses applications contactent Google sans raison justifiée dans le service qu'elles délivrent. Dans [72], les auteurs suggèrent une approche permettant la détection d'applications malveillantes Android au travers d'exécutions en environnement protégé. Cette approche se base sur les appels à l'API réalisés par une application pour construire un modèle markovien de son comportement et se base ensuite sur ce modèle pour identifier les applications malveillantes. Pour permettre la résilience de leur approche aux modifications de l'API Android, ainsi qu'à l'évolution des applications malveillantes, les auteurs de cette proposition ont choisi d'abstraire les appels à l'API. Une fois établi un large ensemble de modèles d'applications, ce système a recours à des méthodes de classification de sorte à identifier les applications malveillantes ou non. Evaluée avec un dataset de 8500 applications bénignes et 35500 applications, cette approche a permis la détection de 99% des virus contenus dans le dataset et a conservé sa précision jusqu'à deux ans après l'exécution de la première série d'expérimentations. Enfin les travaux présentés dans [101] font appel à la génération automatique de descriptions orientées vers la sécurité d'applications Android. Ces travaux se basent sur l'observation déjà évoquée que le système de permissions Android n'est pas facilement compréhensible et que ne fournir que les permissions déclarées par une application ainsi qu'une brève description textuelle fournie par le développeur n'est pas suffisant pour éclairer le choix des utilisateurs. Cette approche propose donc la génération automatique de descriptions orientées vers la sécurité des applications Android en complément aux deux approches précédemment citées. L'évaluation de ces travaux en révèlent le bénéfice pour les utilisateurs d'applications Android. Toutefois comme les deux approches précédentes ce travail ne peut s'appliquer que de manière proactive et ne permet donc pas la protection d'utilisateurs en réaction à l'installation d'une application malveillante.

Une approche visant l'établissement de profils réseaux d'applications Android a été publiée dans [32]. Ce travail se base sur l'observation que les applications Android sont difficiles à spécifiquement repérer dans le trafic géré par les réseaux, puisqu'elles communiquent classiquement par HTTP ou HTTPS. Les auteurs de cette approche s'appuient une fois de plus sur une exécution en environnement protégé pour collecter des traces du trafic généré par une application afin de construire sa signature. Cette signature peut ensuite être utilisée dans des réseaux réels pour détecter l'application. Evaluée avec des milliers d'applications Android, cette approche a démontré son efficacité pour améliorer leur détection, il reste toutefois à déplorer que cette ap-

Approches	Niveau	Cible	Domaine d'analyse	Dataset	Profilage
[67]	Réseau	DoS, port scan, worm, botnet	Non	Non	Non
[98]	Réseau	Port scan	Flux réseau	Non	Réactif
[68]	Réseau	-	Flux réseau	Trafic sain	Non
[49]	Réseau	Botnet	Flux réseau	Trafic de botnets	Préventif
[32]	réseau	Applications malveillantes	Flux réseau	Non	Préventif
[39]	Système et réseau	Applications malveillantes	Non	Non	Non
[51]	Perm. et système	Applications malveillantes	Descriptifs	Non	Préventif
[99]	Réseau, système et perm.	Fuite de données	Permissions, flux et logs sys.	Non	Préventif
[10]	Perm. et système	Permissions confuses	API Android	Non	Préventif
[81]	Perm. et système	Sur-privilèges	Manifest et binaire	Non	Préventif
[7]	Perm.	Collusion d'applications	Manifests d'applications	Non	Préventif
[101]	Système	Mauvaises descriptions d'apps	binaire d'apps	Non	Préventif
[72]	Système	Applications malveillantes	Appels système	Non	Préventif
[100]	Système	Fuite de données	appels à l'API	Non	Réactif
[83]	Réseau	Fuite de données	Flux réseau	Non	Réactif
[64]	Système et réseau	Apprentissage de protocoles	Binaire	Non	Préventif

FIGURE 2.1 – Tableau de synthèse des approches relatifs à la sécurité des environnements intelligents (protection des applications Android)

proche ne propose pas de réponse dédiée à déployer en ce sens. Dans [100], les auteurs indiquent une approche permettant le profilage d'applications Android en temps réel. Cette approche se base sur une synergie entre analyse statique et analyse dynamique pour détecter d'éventuelles fuites de données pratiquées par une application Android. Evaluée avec un dataset comportant 1000 virus connus et 400 applications réelles cette approche a permis la détection effective des virus tout en étant 8,3 fois plus rapide que les approches connues en utilisant 90% de mémoire de moins. Ces performances rendent cette approche utilisable tant par les gestionnaires des markets que par les utilisateurs, toutefois il reste à déplorer qu'aucune autre attaque que les fuites de données ne soit prise en compte. Les travaux présentés dans [64] permettent l'établissement du protocole utilisé par une application Android. Cette approche se base sur le binaire d'une application pour reconstituer le protocole de ses communications réseau, il s'agit donc une fois de plus d'une approche préventive et non réactive. A partir de l'exécutable d'une application le système proposé dans cette approche identifie les paires de requêtes et réponses HTTP ainsi que leurs données liées. Outre la limite déjà citée (que cette approche n'est pas exploitable en réaction à l'installation d'une application malveillante), il est à signaler que cette approche ne propose pas la mise en place automatisée d'une solution permettant la protection des utilisateurs d'Android.

2.1.4 Synthèse des travaux existants et limites

La synthèse des travaux relatifs à la protection d'environnements intelligents se trouve dans le tableau 2.1. Il ressort des travaux que nous avons examinés que peu d'entre eux considèrent la sécurité à plusieurs niveaux différents (réseau, système et permissions). En outre la très grande majorité des travaux visent à assurer une détection proactive des applications malveillantes, en revanche peu d'entres elles appréhendent le problème d'un point de vue réactif au cas où l'utilisateur aurait déjà installé une application malveillante. Les travaux touchant à la protection des applications au niveau réseau demandent de s'appuyer sur des datasets dédiés, néanmoins il n'existe que peu de semblables datasets. Dans le chapitre 4, nous présenterons comment nous exploitons à la fois les informations issues de l'observation du comportement réseau d'applications et des permissions déclarées par leur fichier de manifest.

2.2 Programmabilité des réseaux pour la sécurité

La protection des équipements intelligents peut s'appuyer sur l'usage de chaînes de fonctions de sécurité déployées en environnements cloud. Ces chaînes reposent sur les approches de réseaux programmables, en particulier de *Software Defined Networking* (SDN) ainsi que sur les solutions de virtualisation de fonctions réseau (NFV, *Network Function Virtualization*). Nous présentons ici les approches relatives à ces différents domaines, nous commençons par les réseaux programmables SDN, qui forment la base de notre stratégie d'orchestration de chaînes de fonctions de sécurité. Après cela, nous développons plusieurs travaux relatifs ayant trait au développement de langages de programmation de contrôleurs SDN, qui facilitent leur utilisation. Enfin, nous présentons les travaux concernant les méthodes de virtualisation de fonctions réseau, qui peuvent être utilisées conjointement aux réseaux SDN pour le déploiement de fonctions de sécurité.

2.2.1 Réseaux programmables SDN

La méthodologie propre aux chaînes de fonctions de sécurité repose essentiellement sur le paradigme de réseaux programmables SDN présentée dans la figure 2.2. Ce nouveau paradigme

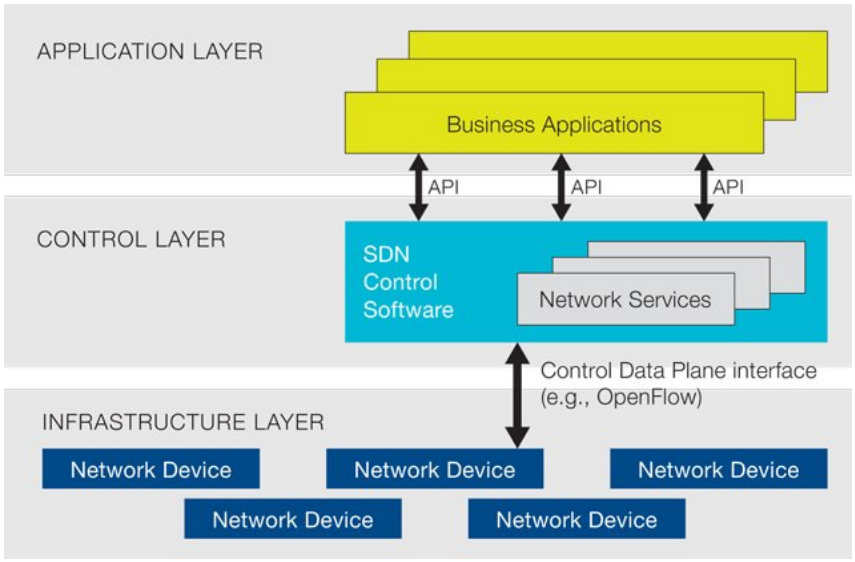

FIGURE 2.2 – Schéma de l'architecture SDN (www.opennetworking.org)

réseau repose sur la distinction de deux plans, le plan de données et le plan de contrôle. Le premier de ces deux plans est représenté par les équipements chargés de la transmission des paquets au niveau du réseau, typiquement des commutateurs SDN encore appelés open V switches ou switches programmables. Le plan de contrôle est pour sa part représenté par un composant central, le contrôleur chargé de la reconfiguration des switches programmables en fonction des évènements affectant le réseau. Les approches de réseaux programmables SDN sont notamment le fruit d'un long travail de recherche présenté dans [42]. La motivation première était d'introduire davantage de flexibilité dans la configuration des réseaux car en effet les opérateurs en charge de leur configuration doivent composer manuellement avec le déploiement de multiples équipements souvent propriétaires. En réponse à cette problématique des chercheurs académiques ont proposé le concept de réseaux actifs permettant la configuration logicielle d'un réseau à partir d'un seul point centralisé, c'est cette intuition qui a conduit au développement du paradigme SDN. Bien que permettant une plus grande flexibilité des réseaux il est toutefois à signaler que ces méthodes introduisent une plus grande complexité d'administration qui peut mettre en difficulté l'administration des réseaux telle que nous la connaissons aujourd'hui. Le paradigme SDN se base sur la notion de flux réseau pour représenter le trafic à analyser. Comme il a déjà été discuté dans la section précédente l'utilisation des flux permet de réduire la charge du trafic total à analyser, en revanche elle ne permet pas de procéder à une analyse profonde du contenu des paquets. Cette limitation peut néanmoins se justifier par le fait qu'une grande partie du trafic est chiffré dans les réseaux modernes et que l'analyse du contenu des paquets en est ainsi rendu plus ardue. En définitive la vision du trafic basée sur les flux dans le paradigme SDN représente un premier stade de traitement auquel on peut faire suivre une phase d'analyse profonde des paquets en fonction de certains critères qui seront discutés au chapitre 4. La communication entre les plans de contrôle et de données au sein de l'architecture SDN repose sur un protocole dédié, typiquement le protocole OpenFlow [73]. Il repose sur des switches ethernet disposant d'un canal sécurisé au travers duquel les contacter. La table de flux de ces switches contient des entrées correspondant à des règles associées à chaque flux pouvant être détecté au sein du réseau. Chacune de ces entrées est caractérisée par trois champs : un champ d'entête décrivant le trafic accepté par la règle ; une action associée à ce type de trafic ; un compteur enregistrant

des statistiques quant à l'activation de la règle. Il est néanmoins à déplorer que le caractère bas niveau des règles OpenFlow rend ardues la manipulation et la validation de celles-ci.

Les cas d'usage des applications de SDN sont multiples, incluant le domaine de la sécurité au sein des réseaux [48]. La rapidité de traitement des switches SDN permet d'envisager le déploiement de certaines fonctions de sécurité, telles que par exemple des pare-feux ou des systèmes de monitoring directement à ce niveau. Néanmoins, la configuration de chaque entrée pour chaque switch du réseau induit une granularité très fine augmentant la probabilité d'introduire des règles contradictoires, par exemple des règles autorisant et bloquant le trafic à destination d'un même numéro de port. Dans le paradigme SDN, les flux sont analysés en première instance par les switches programmables du plan de données. Le trafic entrant dans un switch est successivement analysé par toutes les règles jusqu'à en trouver une qui corresponde, si aucune règle n'est identifiée alors une action par défaut est exécutée. Les règles supportées par les switches SDN peuvent spécifier qu'un flux ou un paquet doit être transmis au contrôleur pour faire l'objet d'analyses plus profondes. Le contrôleur d'un réseau est en effet en charge d'analyser l'état du réseau, afin de identifier des évènements anormaux nécessitant une reconfiguration des switches programmables. Pour ce faire, le contrôleur peut premièrement être alerté par un switch comme il a déjà été discuté, il peut encore collecter des informations statistiques sur l'activation des règles déployées dans le réseau sur la base des compteurs d'activation qui y sont associés. Un exemple d'évènement demandant la reconfiguration des règles d'un réseau SDN est la découverte d'un hôte infecté par un virus et devant être isolé du reste du réseau afin de limiter le risque de contagion. Dans le cas où l'état du réseau nécessite une telle reconfiguration alors le contrôleur doit calculer la modification requise et l'appliquer sur les switches programmables. Le calcul de cette reconfiguration est défini par l'opérateur en charge d'un réseau afin de en automatiser la gestion, d'une certaine manière le paradigme SDN permet d'abstraire les tâches de gestion matérielle du réseau derrière une interface logicielle de la même manière qu'un système d'exploitation cache les aspects matériels de la gestion de la mémoire dans un ordinateur. L'un des rôles du contrôleur dans un réseau SDN est d'ailleurs d'offrir des interfaces de programmation de haut niveau à des applications qui les utiliseront de la même manière que les applications déployées sur un ordinateur font appel au matériel au travers des fonctions mises en place par le système d'exploitation. Ces applications peuvent avoir différents rôles au sein du réseau, elles peuvent en optimiser la gestion, jouer un rôle sur la sécurité ou encore assurer la distribution de calculs complexes entre plusieurs hôtes. La programmation d'un contrôleur SDN peut être grandement facilitée par l'utilisation de langages dédiés.

2.2.2 Langages pour la programmation de contrôleurs SDN

De nombreux langages pour la programmation de contrôleurs SDN ont été proposés en vue d'en simplifier la configuration. On peut notamment citer le langage Procera [41] qui a été spécifié dans le but de permettre aux opérateurs réseau d'implémenter des fonctions de gestion de haut niveau tout en abstrayant la gestion concrète des règles OpenFlow. Ce langage a été proposé pour la gestion de plusieurs types de réseau différents, notamment des réseaux universitaires ou des réseaux d'industries pour lesquels il a permis une facilitation des tâches de gestion et de protection. S'il permet le développement facilité de fonctions d'administration complexes ce langage souffre toutefois de la limitation qu'il n'apporte pas de moyen d'en valider la correction, par exemple au travers de méthodes de vérification formelle. Citons également le langage Pyretic [44] faisant partie de la famille de langages de programmation développés par Nate Foster et Jennifer Rexford. Ce langage de programmation, implémenté en Python, permet la spécification du comportement devant être adopté par le plan de données pour tous types de

trafic accepté par le réseau. Pour ce faire Pyretic repose sur des politiques de base permettant de définir le comportement à associer au trafic entrant dans le réseau, ces éléments de base pouvant ensuite être composés afin de former des politiques plus abouties. Pour permettre une meilleure compréhension des travaux qui seront présentés au chapitre 4 nous présentons ici les politiques de base mises à disposition par le langage Pyretic :

- *identity* : cette règle transmet tous les paquets entrants ;
- *drop* : cette règle supprime tous les paquets entrants ;
- *match*$(x_1 = y_1, \ldots, x_n = y_n)$: cette règle transmet les paquets dont les champs d'entête x_i contiennent les valeurs y_i ;
- *modify*$(x_1 = y_1, \ldots, x_n = y_n)$: cette règle transmet tous les paquets en modifiant leur champs d'entête x_i aux valeurs y_i ;
- *query* : cette règle représente la transmission d'un paquet au contrôleur pour de plus amples analyses ;
- *countPackets*$(x_1 = y_1, \ldots, x_n = y_n)$: cette règle compte le nombre de paquets dont les champs d'entête x_i contient la valeur y_i ;
- *limitFilters*$(k, x_1 = y_1, \ldots, x_n = y_n)$: cette règle transmet un maximum de k paquets dont les champs d'entête x_i contient la valeur y_i ;
- *regexpQuery*(*pattern*) : cette règle transfère le trafic dont le payload correspond à l'expression régulière fournie en paramètre à regular expression).

Ces diverses règles de base peuvent ensuite être combinées au moyen de divers opérateurs de composition. Pyretic supporte trois opérateurs intéressants dans notre contexte : l'opérateur de composition séquentielle $\gg$; l'opérateur de composition parallèle $+$; l'opérateur de négation $\sim$. Considérant deux politiques p_1 et p_2 leur composition séquentielle $p_1 \gg p_2$ renvoie tous les paquets acceptés à la fois par p_1 et p_2 sous condition que les modifications apposées par p_1 soient acceptées par p_2. Leur composition parallèle $p_1 + p_2$ renverra tous les paquets acceptés par p_1 ou par p_2 et enfin $\sim p_1$ la négation de p_1 qui renverra tous les paquets rejetés par p_1.

Pyretic est également rendu intéressant par son extension Kinetic [63] qui permet la vérification du plan de contrôle spécifié sous la forme d'un automate, où le réseau peut passer de la politique identity à drop sur la base de la détection d'une infection. Les utilisateurs de Kinetic peuvent compléter cette spécification de leur contrôleur avec la déclaration de propriétés exprimées dans la logique temporelle CTL (Computation Tree Logic), par exemple pour la politique que nous avons considérée la propriété $\mathbf{E}\,\mathbf{F}$ *infected* exprime le fait qu'il existe un chemin d'exécution tel que le système entre dans l'état infecté. Kinetic permet de particulariser de tels automates pour chaque connexion du réseau sur la base d'un mécanisme associant un automate à certains champs d'entête. Ces travaux manquent toutefois d'approfondissements en ce qui concerne la vérification du plan de données. Nous y reviendrons au chapitre 5.

(1) edge node $\neg$*infected* (0)

Il convient également de citer le langage Frenetic [45] pour lequel nous avons eu un intérêt tardif dans le cadre de notre travail. Pyretic n'est en effet plus supporté par les contrôleurs actuels, c'est Frenetic qui a donc servi pour le déploiement de nos travaux. Ce langage s'appuie sur des politiques relativement équivalentes à celles proposées par Pyretic pour spécifier des politiques de sécurité. Ces politiques sont spécifiées sous la forme de prédicats tels que par exemple *IP4SrcEq* qui spécifie que l'adresse source d'un paquet doit avoir une certaine valeur ou encore *TCPDstPortEq* qui spécifie que le port TCP à destination d'un paquet doit avoir une valeur donnée. Ces prédicats peuvent être combinés au moyen d'opérateurs logiques $\wedge$, $\vee$ et $\neg$ puis intégrés dans une politique *Filter* afin d'en répercuter les effets sur une politique spécifiée par un utilisateur. Une politique *Filter* du langage Frenetic doit ensuite être associée avec d'autres pour permettre la spécification de règles de configuration valide. Frenetic supporte

plusieurs règles telles que par exemple *SetPort* qui indique sur quel port ethernet d'un switch SDN le trafic doit être émis ou encore *SetTCPDstPort* qui modifie la valeur du port TCP d'un paquet donné. Ces politiques peuvent être combinés au moyen d'opérateurs analogues à ceux mis à disposition par le langage Pyretic, $\gg$ tient toujours lieu d'opérateur de composition séquentielle, | joue ici le rôle de la composition parallèle et $\sim$ sert de nouveau de négation. Une fois spécifiée en Python une politique Frenetic doit être traduite en JSON avant d'être transmise au contrôleur dédié. Ce format intermédiaire, appelé netkat par les développeurs de Frenetic, sera interprété par le contrôleur afin de générer les règles OpenFlow correspondantes. Une fois cela fait les règles sont placées sur les switches correspondants dans le réseau. Pour permettre le placement de règles Frenetic met à disposition de ses utilisateurs le prédicat *SwitchEq* qui permet d'indiquer sur quel switch une règle doit être déployée. Si aucun switch n'est indiqué dans un *Filter* alors les règles concernées sont par défaut déployées sur tous les switches du réseau. Le plan de contrôle d'une politique Frenetic est géré au moyen d'une classe Python dédiée. Cette classe supporte différentes méthodes telles que *connected* ou *switch_up* qui caractérisent les différents évènements pouvant survenir pendant l'exécution du réseau. Lorsqu'un tel évènement survient alors le contrôleur attaché à Frenetic le transmet à l'application concernée qui exécute alors l'action correspondante. En fonction du type d'évènement reçu l'application peut alors procéder à une mise à jour du plan de données, à l'émission de trafic sur un switch donné ou encore se livrer à toute tâche de gestion propre à la maintenance du réseau.

2.2.3 Virtualisation de fonctions de sécurité

Les approches de virtualisation peuvent être utilisées en réponse aux différents enjeux de sécurité évoqués dans la section précédente. En particulier on peut citer les travaux liés aux middleboxes, aux réseaux programmables tels que SDN, au chaînage de fonctions de sécurité et à l'application de ces diverses approches à la protection des environnements intelligents. Comme brièvement expliqué en introduction, nous appelons *Network Function Virtualization* (NFV) le paradigme relatif aux méthodes de virtualisation de fonctions réseaux, et *Virtual Network Function* (VNF) les instances de fonction réseau virtualisées, par exemple des pare-feux ou des systèmes de détection d'intrusion. Lesdites solutions peuvent également être déployées au moyen de middleboxes qui sont des équipements typiquement matériels permettant d'assurer les mêmes services réseau. Conformément à ce qu'indique le RFC 3234 [23] une middlebox est un équipement intermédiaire permettant d'exécuter d'autres fonctions que celles normalement exécutées par les routeurs au sein du paradigme IP. Les fonctions assurées par les middleboxes s'étendent du champ de la sécurité informatique jusqu'à la gestion des réseaux. Pour citer quelques exemples de middleboxes utilisées à fin de sécurité mentionnons les pare-feux ou les systèmes de détection d'intrusion. Pour les middleboxes ayant fin de gestion on peut citer les load balancers ou encore les Network Address Translater (NAT). Bien qu'offrant de nombreux services les middleboxes apportent également nombre de difficultés ayant fait l'objet de plusieurs travaux de recherche présentés dans cette section.

La première difficulté inhérente à l'utilisation des middleboxes est que l'on n'a pas encore pu en réaliser un modèle générique bien que plusieurs travaux aient été menés en ce sens. Ainsi, dans [38], les auteurs proposent un modèle générique basé sur un langage dédié. Ce langage permet d'exprimer des propriétés telles que la pré et la post condition d'une middlebox ainsi que le type de trafic qu'elle peut accepter. Bien qu'il permette la spécification de modèles très détaillés de middleboxes ce travail souffre du fait qu'il n'est pas rattaché à une implémentation pratique des services qu'il décrit. En outre, la diversité et la complexité d'administration propres au grand nombre de middleboxes existant sur les marchés rendent délicate la spécification d'un modèle

générique de leur comportement sans pour autant perdre de la sémantique propre à chaque équipement. Dans [55], les auteurs proposent une autre approche permettant la spécification de pare-feux dédiés à la protection de réseaux SDN. L'article propose une approche intéressante permettant la résolution de conflits entre les règles déployées dans le réseau et leur ajustement dynamique grâce à la flexibilité introduite par le paradigme SDN. Bien qu'il soit implémenté et exploitable en pratique la limite de ce travail est qu'il se cantonne à la représentation de pare-feux et ne peut donc pas être généralisée à d'autres middleboxes. En résumé la limite des travaux sur la modélisation de middleboxes est qu'il n'existe pas encore de modèle qui soit à la fois générique et exploitable en pratique. Un second problème relatif à la gestion de middleboxes est la difficulté relative à leur configuration et à leur déploiement. Plusieurs travaux s'attachent d'ailleurs à introduire des stratégies sensées faciliter ces tâches en environnement d'exploitation. Ainsi dans [9] les auteurs proposent d'utiliser les approches de réseaux programmables SDN pour simplifier la gestion de middleboxes. Cette approche part du plan de middleboxes à déployer et le traduit sous la forme de règles OpenFlow devant ensuite être déployées par le contrôleur du réseau SDN. Ces travaux souffrent néanmoins du sur-coût inhérent à l'utilisation des réseaux SDN et s'ils simplifient le déploiement de middleboxes ils complexifient en revanche la gestion du réseau en général. Dans [95], les auteurs proposent d'externaliser le déploiement de middleboxes en environnement cloud au travers de méthodes de virtualisation, néanmoins ces travaux reportent la complexité d'administration du côté des fournisseurs de réseau qui doivent alors gérer le surcoût lié à la gestion des middleboxes. En définitive, si les solutions de middleboxes permettent une plus grande flexibilité dans la protection des réseaux, elles introduisent également des défis du fait de la complexité de leurs interfaces et souffrent de l'inconvénient qu'il n'existe pas d'interface générique pour leur manipulation.

2.2.4 Chaînage de fonctions de sécurité

Les travaux dans le champ des middleboxes et des réseaux programmables SDN ont donné lieu à de nombreux travaux portant sur le chaînage de fonctions de sécurité comme indiqué dans [22]. L'utilisation des méthodes de virtualisation conjointement avec la programmabilité introduite par les réseaux SDN peut en effet permettre le déploiement de politiques de sécurité flexibles et adaptables en fonction des besoins des utilisateurs. Toutefois du fait de la très forte complexité d'administration déjà discuté relativement à ces diverses approches, il reste nécessaire de considérer l'intégration de solutions de validation empruntées au champ des méthodes formelles pour en assurer la correction avant le déploiement. Certains travaux ont été proposés pour composer directement les réseaux SDN en eux-mêmes tels que par exemple dans [47]. Les auteurs de cette approche proposent de spécifier les fonctions nécessaires à la gestion et à la sécurisation d'un réseau à un haut niveau d'abstraction avant de les composer au travers d'opérateurs semblables à ceux de Pyretic et Frenetic. Les chaînes ainsi obtenues sont ensuite compilées en règles OpenFlow avant d'être déployées dans le réseau. Bien qu'elle permette la spécification et le déploiement de fonctions de sécurité à un haut niveau d'abstraction, cette approche souffre toutefois de la limite induite par l'absence de méthodes de validation pour assurer la correction des chaînes avant leur déploiement. Enfin, on peut citer les travaux présentés dans [79] qui proposent l'utilisation de méthodes de virtualisation pour le chaînage automatique de fonctions de sécurité. Cette approche se propose de résoudre le problème d'allocation des ressources propre aux méthodes de virtualisation, en particulier il propose une construction des chaînes divisée en plusieurs étapes, d'abord la composition, ensuite l'instanciation et finalement le déploiement dans le réseau. Cette approche introduit ainsi une plus grande flexibilité dans la gestion du réseau, toutefois la complexité d'administration déjà mentionnée des approches de

Approches	Finalité	Complexité	Modélisation	Abstraction	Vérifiabilité
[42]	Administration	Forte	Forte	Faible	Faible
[73]	Administration	Forte	Forte	Faible	Faible
[48]	Sécurité	Forte	Forte	Faible	Faible
[41]	Administration	Modérée	Forte	Forte	Faible
[44]	Administration	Modérée	Forte	Forte	Forte
[45]	Administration	Modérée	Forte	Forte	Forte
[63]	Administration	Modérée	Forte	Forte	Forte
[38]	Sécurité	Modérée	Modérée	Forte	Faible
[55]	Sécurité	Modérée	Forte	Forte	Modérée
[9]	Administration	Forte	Forte	Forte	Faible
[95]	Sécurité	Forte	Forte	Modérée	Faible
[22]	Sécurité	Forte	Forte	Faible	Faible
[47]	Sécurité	Forte	Forte	Forte	Faible
[82]	Sécurité	Forte	Forte	Forte	Forte
[79]	Sécurité	Forte	Forte	Forte	Faible
[87]	Sécurité	Forte	Forte	Forte	Faible
[78]	Sécurité	Forte	Forte	Forte	Faible
[62]	Sécurité	Forte	Forte	Faible	Faible
[57]	Sécurité	Forte	Forte	Faible	Faible
[56]	Sécurité	Forte	Forte	Faible	Faible

FIGURE 2.3 – Tableau de synthèse des travaux sur la programmabilité réseau et la virtualisation de la sécurité dans le réseau

virtualisation constitue un solide obstacle à son utilisation pratique. Certains travaux sur la composition de fonctions de sécurité sortent du champ d'applications de SDN, c'est notamment le cas des travaux présentés dans [82] qui proposent de s'assurer automatiquement de la cohérence de politiques réseau lors de leur construction. Pour ce faire, l'approche préconisée par les auteurs du papier consiste à construire un graphe de la politique réseau devant être déployé, ce graph étant validé automatiquement au moyen d'opérations algébriques sur sa structure. La politique de sécurité résultante est ensuite compilée en règles de configuration de bas niveau avant d'être déployée dans le réseau. Bien qu'ils permettent une plus grande sûreté des politiques déployées dans le réseau, ces travaux présentent toutefois une certaine lourdeur de traitement qui en rend difficile l'application pratique.

Les diverses approches présentées dans cette section peuvent être utilisées comme des solutions aux problèmes de sécurité liés aux environnements intelligents. Dans [87] les auteurs suggèrent de développer des politiques de sécurité dédiées à chaque utilisateur au travers des méthodes de virtualisation. Une telle approche permet de résoudre la problématique déjà mentionnée de la limitation des équipements intelligents en termes de batterie ainsi que de ressources CPU, elle permet en outre d'assurer la flexibilité requise par le domaine pour assurer la protection d'équipements mobiles évoluant rapidement. Néanmoins elle présente la problématique d'introduire une grande complexité d'administration dans le réseau du fait de sa très forte granularité. Dans [78], les auteurs indiquent une seconde approche permettant la protection d'environnements intelligents au travers du déploiement d'antivirus dans le cloud. Cette approche vise à réduire le coût du déploiement d'antivirus directement sur les équipements intelligents. Evaluée dans le contexte d'équipements Nokia cette approche a démontré ses capacités en termes de minimisation du coût induit sur les équipements, on peut faire à ce travail la critique de se can-

tonner à des antivirus et de ne pas envisager le déploiement d'autres fonctions de sécurité telles que par exemple des pare-feux ou des systèmes de détection d'intrusion. Dans [62], l'approche permet la mise en place de pare-feux pour des équipements Android au travers de méthodes de virtualisation. Ce travail s'appuie sur la spécification des besoins spécifiques d'utilisateurs du système d'exploitation pour spécifier dynamiquement la configuration de sécurité visant à assurer leur protection. Cette approche permet elle aussi de résoudre le problème de limitation des ressources des équipements Android, toutefois elle se cantonne à la spécification de pare-feux et n'appréhende pas d'autres fonctions de sécurité. On peut en outre lui faire la critique d'introduire une forte complexité d'administration dans le réseau sans pour autant mettre en place de dispositif de validation tel que par exemple par application de méthodes formelles. D'autres travaux ont été publiés pour appliquer les chaînes de fonctions de sécurité à la protection d'environnements intelligents. C'est notamment le cas de l'approche présentée dans [57] pour appliquer des chaînes construites dans le cloud à la protection d'environnements Android. Cette approche se base sur le comportement réseau observé pour une application pour en déduire la chaîne de fonctions de sécurité la plus adaptée, néanmoins ce travail souffre de la limitation que le chaînage est construit de manière manuelle et non systématique. En outre le modèle de fonctions de sécurité qui est ici adopté ne permet pas de généralisation à de nouveaux types de fonctions tel que par exemple des systèmes d'inspection profonde. Enfin, dans [56] est décrit le prolongement des travaux précédents vers le déploiement de chaînes spécifiques au comportement réseau et aux permissions observées pour une application. Ce travail enrichit le précédent par l'introduction des permissions qui permettent d'appréhender un modèle des données transmises par une application, cette information permettant d'inférer des règles pour la configuration d'un système de prévention de fuite de données. Toutefois cette approche, comme plusieurs autres présentées dans cette section, introduit une forte complexité d'administration dans le réseau qui nécessite d'avoir recours à des méthodes de vérification formelle pour en valider la correction avant le déploiement.

2.2.5 Synthèse des travaux existants et limites

Le tableau 2.3 fait la synthèse des travaux touchant à la programmabilité réseau et à la virtualisation de la sécurité au sein du réseau. De nombreux travaux touchent à l'automatisation de l'administration et de la sécurisation de réseaux au travers du paradigme SDN. Toutefois, ils introduisent une forte complexité d'administration dans le réseau. Pour remédier à ce problème, plusieurs langages de haut niveau ont été proposés pour simplifier la spécification de politiques, tout en permettant l'utilisation de méthodes de vérification formelle pour en valider la cohérence avant le déploiement. Ces travaux peuvent être utilisés conjointement avec les méthodes de virtualisation pour déployer automatiquement des fonctions de sécurité virtuelles, néanmoins la limitation affectant ces approches est l'absence d'un modèle générique desdites fonctions de sécurité, ainsi que la complexité d'administration qu'elles introduisent dans le réseau. Enfin, plusieurs travaux ont été proposés pour permettre le chaînage de fonctions de sécurité, notamment en vue de la protection d'environnements intelligents. La principale limitation en est encore une fois la forte complexité d'administration introduite dans le réseau.

2.3 Méthodes de vérification formelle

La grande complexité d'administration des chaînes de fonctions de sécurité et des règles de configuration des réseaux SDN qui les sous-tendent obligent à considérer l'utilisation de méthodes de vérification formelle pour en valider la correction. Dans le cadre de notre travail, nous

nous sommes intéressés aux méthodes de model checking, de SMT solving et de programmation linéaire brièvement présentées dans cette section. Nous détaillons ensuite plusieurs familles d'applications de méthodes formelles ayant intéressé notre travail, en particulier la vérification formelle de politiques de sécurité réseau, l'apprentissage de processus et enfin la résolution de problèmes d'optimisation basée sur les méthodes formelles.

2.3.1 Méthodes de model checking

Dans notre travail nous avons fait appel à plusieurs types de méthodes formelles, en premier lieu les méthodes de model checking. Ces méthodes, introduites par Emerson et Clarke [30] ont originellement été appliquées à la vérification de systèmes parallèles et distribués impliquant des accès concurrents à certaines ressources, que ce soit par une mémoire partagée ou par des communication réseau. Les erreurs de ce type de programme sont particulièrement difficiles à détecter par tests unitaires en raison de la nature éminemment parallèle des systèmes qu'elles affectent. L'introduction des méthodes de model checking permet une prise en compte de cette problématique en abordant les systèmes parallèles au point de vue de leur interaction et plus de leurs composants isolés. Le problème de model checking est défini ainsi dans [30] : étant donné M une structure de Kripke, c'est-à-dire un système d'états et de transitions et étant donnée f, une formule logique, vérifions que M est un modèle de f, c'est-à-dire que f est vraie pour la structure M et se note $M \vdash f$. Les états de la structure M représentent la valeur de ses variables à un point particulier et ses transitions représentent les pas de calcul. Il est en outre possible de combiner plusieurs structures de Kripke au moyen d'opérateurs de produit introduisant néanmoins une complexité algorithmique supplémentaire au problème. Pour gérer cette complexité algorithmique additionnelle, la plupart des model checkers modernes proposent différents algorithmes tels que le model checking borné qui cantonne l'évaluation du modèle aux états atteignables en un nombre n de transitions, la valeur de n restant à la discrétion des utilisateurs.

Une extension probabiliste du problème de model checking a été introduite dans [43] afin de permettre la vérification de systèmes stochastiques. Ces systèmes peuvent ainsi être représentés sous la forme de chaînes de Markov discrètes ou continues ou encore sous la forme de processus de décision markoviens. Cette extension permet de quantifier la probabilité qu'un système se trouve dans un état s donné, ce qui présente un réel intérêt dans le cas de la validation de systèmes non-déterministes. Il est toutefois à noter que l'intégration du paramètre probabiliste n'est pas sans impact, la complexité algorithmique des méthodes de model checking probabiliste étant sensiblement plus élevée que celle des algorithmes traditionnels. Le problème de model checking peut s'appréhender au travers de différentes logiques outre la logique propositionnelle. Ainsi dans [97] E.M. Clarke et A. P. Sistla abordent le problème de model checking pour la logique temporelle linéaire ou Linear Temporal Logic (LTL) dans laquelle les propriétés des systèmes sont proposées sous forme d'une structure logique du temps. Un exemple d'une telle formule est **G F** *critical* où les opérateurs **G** pour globally et **F** pour finally traduisent le fait que le système sera toujours finalement en état critique. LTL ne permet cependant pas d'envisager plusieurs avenirs possibles. Dans [75] les auteurs discutent l'utilisation de logiques de branchement telles que Computational Tree Logic (CTL) dans laquelle les propriétés temporelles des systèmes sont perçues avec plusieurs avenirs possibles. Ces avenirs sont exprimés grâce aux opérateurs de quantification de chemin universel **A** et existentiel **E**. Pour reprendre l'exemple précédent la formule **A F** *critical* traduit le fait que pour tous les chemins d'exécution le système sera finalement en état critique, la principale différence entre CTL et LTL réside dans le fait que la première envisage les états du système alors que la seconde envisage leur succession. Il est

toutefois à noter que CTL ne permet pas l'accumulation d'opérateurs temporels comme on a pu le voir avec LTL. La logique CTL∗ [29] a été proposée pour répondre aux limitations des deux précédentes logiques. La formule **A G F** *critical* exprime le fait que le système sera toujours finalement en état critique pour tous les chemins d'exécution.

La principale limitation de toutes ces méthodes de model checking tient à leur complexité algorithmique. Comme indiqué dans [97] la complexité algorithmique du problème de model checking pour LTL est EXPTIME, il en va de même pour CTL∗ d'après [29] ce qui rend ces logiques difficilement utilisables pour des spécifications complexes. CTL est pour sa part affectée d'une complexité algorithmique polynomiale dans la taille du système à vérifier, concernant l'extension probabiliste du problème de model checking il est indiqué dans [43] que la complexité algorithmique des versions probabilistes de LTL, CTL et CTL∗ peut s'élever jusqu'à une exponentielle d'exponentielle.

2.3.2 Méthodes de SMT solving

Une seconde classe de méthodes formelles ayant intéressé notre travail sont les méthodes de SMT solving pour Satisfiability Modulo Theories. Ces méthodes sont le fruit d'une longue maturation autour du problème de satisfiabilité ou encore problème SAT.. Etant donnée une formule booléenne f, le problème SAT consiste à en déterminer soit la satisfiabilité, c'est-à-dire l'existence d'une valuation rendant la spécification vraie, soit la validité, c'est-à-dire la satisfiabilité pour toutes valuations ou encore la non satisfiabilité. Plusieurs algorithmes de résolution ont été proposés pour ce problème, le plus utilisé étant l'algorithme DPLL [37, 36] qui consiste intuitivement à essayer toutes les valuations possibles d'une formule dans l'ordre de ses variables jusqu'à obtenir une contradiction qui l'invalide. L'inconvénient de cette méthode est une fois encore sa forte complexité algorithmique, l'algorithme de DPLL est en effet *NP* Hard, ce qui en rend l'application potentiellement très coûteuse en terme de temps de réponse.

Conformément à [19] le problème SMT est une extension du problème SAT aux théories des entiers, des réels, des mots binaires ou plus récemment des chaînes de caractères [69, 102]. Pour permettre la résolution de problèmes dans ces différentes théories les méthodes de SMT solving s'appuient sur des méthodes dédiées telles que la méthode simplex pour les problèmes arithmétiques [34]. Les SMT solvers supportent également des solvers pour plusieurs théories des structures de données telles que les mots binaires [12], les tableaux ou encore les listes. Pour permettre la spécification de problèmes impliquant plusieurs théories les SMT solvers s'appuient sur SMTlib [8], un langage dédié pour la spécification de problèmes de satisfiabilité impliquant plusieurs théories. Le mode de fonctionnement classique d'un SMT solver est décrit dans la figure 2.4. Une fois chargé, le modèle SMTlib est décomposé en sous-instances relativement à l'algorithme de Nelson et Oppen [13] qui consiste à décomposer un problème suivant les différentes théories qu'il implique. Une fois cette décomposition réalisée le SMT solver s'en remet à l'algorithme de DPLL implémenté par le raisonneur sat pour calculer un modèle d'une version du problème n'impliquant que des variables booléennes en lieu et place des contraintes relatives aux diverses théories qui le composent. Une fois construit ce modèle est transmis aux différents raisonneurs relatifs aux théories impliquées par le problème afin de valider que le modèle booléen est toujours admissible au vu des théories impliquées. Si tel n'est pas le cas alors on recherche un nouveau modèle de la formule via DPLL et on réitère la vérification via les theory solvers jusqu'à trouver un modèle du problème ou à épuiser les modèles booléens, auquel cas on conclut à la non satisfiabilité du problème initial.

Le premier cas d'application des méthodes SMT était les problèmes de vérification formelle, le système à valider et ses contraintes sont fournis sous la forme d'une spécification logique et le

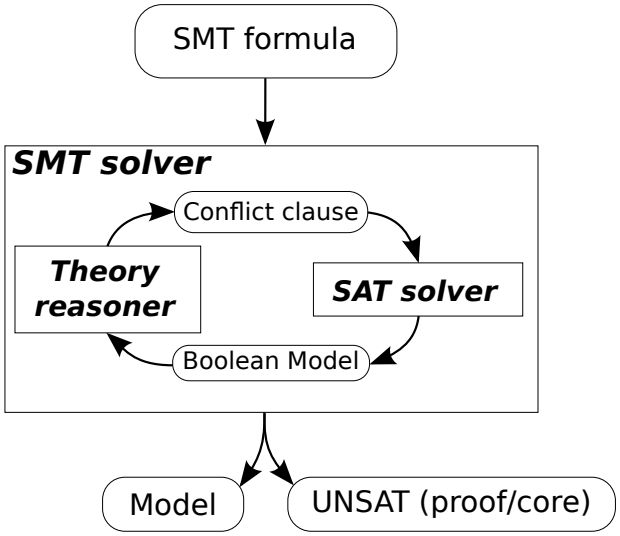

FIGURE 2.4 – Architecture d'un SMT solver

solver renvoie le modèle correspondant si celui-ci existe ou un contr-exemple dans le cas contraire. Il est en outre intéressant de signaler que la plupart des SMT solvers exploitent la dualité des problèmes de satisfiabilité et de validité pour minimiser la complexité de leur implémentation interne. En effet prouver que la formule f est satisfiable est logiquement équivalent à démontrer que $\neg f$ n'est pas valide. Cette dualité permet donc de restreindre les procédures de vérification par SMT solving à des problèmes de satisfiabilité, la validité d'une formule f pouvant être exprimée comme étant la non satisfiabilité de $\neg f$. Parmi d'autres applications, citons la synthèse de programmes guidée par la syntaxe [4] qui permet la génération de programmes ou de fonctions algébriques spécifiées à partir d'un ensemble de contraintes logiques. Les auteurs de [4] proposent de limiter la complexité combinatoire nécessairement liée à une telle approche en minimisant l'espace de recherche par la définition de contraintes syntaxiques, par exemple n'utiliser que les opérateurs $+$, $-$ et la composition conditionnelle *si, alors, sinon*. Cette méthode est supportée par plusieurs SMT solvers modernes tel que par exemple cvc4, toutefois il semble qu'elle ne soit pas encore assez mature pour supporter la synthèse de large programmes tel que des chaînes de fonctions de sécurité.

La principale limitation des méthodes de SMT solving en reste toutefois la complexité algorithmique. Comme nous l'avons déjà abordé le problème SAT est *NP* hard, il en va de même pour la plupart des procédures de décisions relatives aux différentes théories utilisées par les SMT solvers. En définitive si les méthodes de SMT solving sont applicables en pratique il reste toutefois nécessaire d'en limiter la complexité par un minutieux travail de modélisation visant à restreindre la taille des problèmes à résoudre. Ces méthodes sont toutefois très intéressantes dans l'optique de la validation automatique de systèmes par application d'interprétation abstraite, nous y reviendrons au chapitre 5 de ce mémoire.

2.3.3 Méthodes de programmation linéaire

Un troisième groupe de méthodes formelles intéressantes pour notre travail sont les méthodes de programmation logique. Ces méthodes qui s'apparentent à un paradigme de programmation reposent sur des règles d'inférence logique afin de établir la preuve de certaines propositions sur

la base d'un ensemble d'axiomes donnés au système. Ce mode d'inférence repose sur les clauses de Horn [54] pour modéliser les règles logiques : ces clauses comportent au plus un littéral positif. Elles se constituent d'une ou plusieurs prémisses ainsi qu'une conclusion qui peut être omise. Si jamais la conclusion n'est pas spécifiée alors elle est supposée être la valeur de vérité faux, ceci peut être utile pour spécifier des contraintes ou des requêtes sur le système. Ces clauses sont généralement représentées sous la forme de règles de preuves avec les prémisses en numérateur et la conclusion en dénominateur, on peut également les représenter de manière fléchée avec les prémisses pointant vers la conclusion.

Ces méthodes peuvent être utilisées pour la vérification logicielle par application de résolution de contraintes [84]. L'idée est de spécifier un système d'inférence dont les clauses de Horn comportent des variables figurant tant dans leurs prémisses que dans leurs conclusions. A partir de là il est possible de résoudre la valeur des variables de manière syntaxique et de propager cette résolution à l'ensemble du système de clauses jusqu'à atteindre les axiomes ou au contraire une contradiction. Ce mode de raisonnement est au coeur du langage de programmation Prolog [85] qui fera l'objet de développements ultérieurs au chapitre 4. Il est également possible de faire l'application des clauses de Horn pour la preuve de théorèmes mais ceci ne fera pas l'objet du présent mémoire.

2.3.4 Applications de méthodes formelles

Nous abordons ici les travaux relatifs à l'application de méthodes formelles aux diverses problématiques évoquées jusqu'à présent. Nous traiterons en particulier trois familles d'applications intéressantes dans notre contexte, la vérification formelle de chaînes de fonctions de sécurité, l'apprentissage de processus logiciels et enfin les extensions d'outils issus du champ des méthodes formelles aux problématiques d'optimisation.

Vérification de politiques réseau

La vérification formelle de politiques de sécurité a fait l'objet de nombreux travaux de recherche, en particulier dans le domaine émergeant des réseaux SDN. Dans [15], les auteurs proposent une approche visant à la vérification ainsi qu'à la synthèse de configuration du plan de contrôle en vue d'assurer le routage de trafic au sein des réseaux. Cette approche est intéressante du point de vue de la comparaison des méthodes de vérification et de synthèse. Néanmoins le fait qu'elle se concentre sur le routage de trafic l'oriente davantage dans une optique d'administration et pas de sécurisation des réseaux. La détection de comportements malveillants n'y est d'ailleurs pas du tout abordée, ce qui rend cette approche inapplicable dans l'optique de la protection d'équipements intelligents qui nous intéresse. On peut encore citer les travaux relatifs à Vericon [11] qui proposent un framework pour la validation de politiques de routage en réseau SDN intégrant un langage de spécification de propriétés du plan de données basé sur la logique du premier ordre. Ce framework se base sur des méthodes de model checking pour assurer qu'une politique est correcte pour toutes les topologies réseau admissibles et pour toutes les séquences d'évènements potentielles. Les auteurs de ce papier démontrent que leur approche passe à l'échelle de larges topologies par une évaluation pratique en environnement réaliste. On peut toutefois déplorer le fait que ce langage ne fasse pas appel à des logiques temporelles, ce qui en limite le pouvoir d'expression d'attaques impliquant un aspect temporel telles que par exemple des dénis de service ou des scans de port.

Dans [46] les auteurs proposent une approche permettant la vérification automatique de mises à jour du réseau sur la base de model checking. Dans cette approche chaque état du modèle est

identifié à l'état du réseau à un instant donné, les transitions représentent les évènements affectant la gestion du réseau tels que l'émission d'un paquet ou le déploiement de règles. C'est sur ce dernier type d'évènements que se concentre cette approche de vérification, les auteurs proposent de valider qu'une mise à jour n'introduit pas de règles contradictoires dans le réseau. Toutefois on peut leur faire la critique que la granularité de leur modèle en complexifie l'application en situation réelle où de hautes performances sont attendues tant en termes de temps de réponse que de consommation mémoire. Enfin dans [61], les auteurs proposent une approche permettant la vérification d'invariants réseau en temps réel. L'objectif de cette approche est de contourner le problème de complexité des modèles soulevé plus haut en procédant à une vérification non pas globale mais incrémentielle du réseau, en somme plutôt que de vérifier l'intégralité de la configuration les auteurs de cet article proposent de vérifier la validité de chaque règle avant son insertion. Pour réaliser cet objectif les auteurs de cet article attendent des opérateurs du réseau qu'ils spécifient les invariants qu'ils s'attendent à voir garantis dans leur réseau. Toutefois l'aspect très bas niveau de cette approche de vérification la rend difficilement adaptable pour la vérification de larges chaînages de fonctions de sécurité tels que nous les proposons.

L'un des cas d'utilisation les plus fréquemment étudié dans la littérature relative à la vérification de configurations réseau est la vérification de politiques de pare-feux. En particulier dans [3] les auteurs proposent une approche permettant la détection d'anomalies dans de larges politiques de pare-feux distribués par application de méthodes de SMT solving. Plus précisément cette approche vise à repérer des contradictions entre les règles, par exemple l'existence simultanée d'une règle autorisant le trafic à destination d'un numéro de port et d'une règle bloquant le trafic vers ce même numéro de port. Les auteurs de cette approche proposent de corriger de semblables contradictions à partir du contrexemple produit par le solver. Néanmoins on peut déplorer que cette approche ne puisse pas être généralisée à d'autres fonctions de sécurité que les pare-feux et qu'ils n'abordent que les problématiques liées au plan de données. Dans [59] les auteurs proposent une approche permettant la vérification formelle de pare-feux basés sur SDN. Cette approche se base sur l'algèbre de processus pour décrire le comportements de diverses politiques de pare-feux. Ces politiques peuvent ainsi être composées en parallèle afin de valider que leur composition n'introduit pas d'incohérence dans le réseau, il est encore possible de s'assurer à chaque changement de topologie que la politique est toujours viable. Cette vérification est toutefois exécutée à chaque évènement modifiant le réseau, la validation du programme du plan de contrôle en lui-même n'est pas abordée dans ces travaux.

Finalement, un autre cas d'utilisation particulièrement étudié est celui de la validation basée sur les tests. Dans [40], les auteurs proposent une approche permettant la validation du plan de données basé sur des tests unitaires reposant sur un planificateur de tests, un ou plusieurs injecteurs de trafic qui sont implémentés par des hôtes dans le réseau et finalement un module de monitoring et de validation visant à vérifier la correction des tests spécifiés par l'opérateur du réseau. A partir de ces différents modules l'approche des auteurs de cet article est de simuler le scénario spécifié par l'opérateur. Il n'est toutefois pas possible de s'assurer de l'exhaustivité de cette couverture de tests comme c'est le cas avec d'autres approches de vérification formelle. Dans [2], les auteurs proposent une approche permettant la vérification de larges infrastructures OpenFlow fédérées par un ou plusieurs contrôleurs. Leur objectif est d'assurer la détection de règles contradictoires au sein d'un ou de plusieurs switches OpenFlow déployés sous la tutelle d'un ou de plusieurs contrôleurs. Pour ce faire les auteurs de [2] proposent d'encoder la table de flux d'un switch sous la forme d'un diagramme de décision binaire. Une fois effectuée cette traduction ils utilisent les méthodes de model checking symbolique pour valider la correction de l'ensemble de la politique. En cas de découverte d'erreur celle-ci peut être corrigée grâce au contrexemple produit par le model checker qui permet d'illustrer la violation de la politique de

sécurité qui a été observée. Le problème de ces deux approches est qu'elles se concentrent sur la validation du plan de données et laissent de côté celle du plan de contrôle, ouvrant la porte à de potentielles erreurs. Enfin dans [24], les auteurs proposent une nouvelle approche permettant la validation par tests de politiques de sécurité OpenFlow. Ils représentent l'intégralité du réseau en prenant en compte tous les évènements pouvant affecter son exécution tels que par exemple l'envoi d'un paquet, sa suppression ou sa transmission au contrôleur. Ce modèle est ensuite vérifié par application de model checking pour s'assurer de sa cohérence. Toutefois on peut critiquer le choix des auteurs de s'être appuyés sur une version simplifiée des switches SDN pour réduire la complexité de leur modèle, en effet une telle abstraction ouvre la porte à l'introduction d'erreurs qui auraient pu être détectées avec une modélisation plus réaliste.

Apprentissage du comportement d'applications

La construction de modèles réalistes des systèmes sur lesquels appliquer des méthodes formelles nécessite d'avoir recours aux méthodes d'apprentissage de processus logiciels. Ces méthodes se basent essentiellement sur des logs retraçant l'exécution des systèmes, nous en présentons ici l'état de l'art illustrant l'application que nous faisons de ces méthodes dans le cadre de l'apprentissage du comportement d'applications Android qui sera détaillé au chapitre 3. L'une des références dans le domaine des méthodes d'apprentissage de processus est l'algorithme K-tails introduit dans [20]. Cette approche permet d'apprendre le comportement de systèmes logiciels sous la forme d'automates markoviens. Cette approche se base sur la succession d'évènements dans un log pour construire les états de l'automate, le paramètre K indiquant la longueur de cette succession. Bien que faisant référence dans le champ des méthodes d'apprentissage de processus l'algorithme de K-tail tend à produire des modèles très complexes des systèmes à représenter. Il est en outre possible que l'algorithme d'apprentissage en lui-même conduise à une surcharge mémorielle du fait du très grand nombre d'opérations qu'il implique. Cet algorithme a donné lieu à de nombreuses déclinaisons telles que par exemple l'algorithme Synoptic [16] qui permet l'exploration de fichiers de logs au moyen de l'apprentissage d'un modèle markovien des systèmes dont ils retracent l'exécution. Pour permettre le chargement des fichiers de logs Synoptic s'appuie sur des expressions régulières qui permettent d'exprimer la syntaxe d'un évènement dans le fichier source. Une fois la liste d'évènements chargée Synoptic calcule les invariants temporels qui la caractérisent, par exemple les lectures fichiers sont toujours pratiquées après l'ouverture d'un fichier. Une fois calculée cette liste d'invariants Synoptic construit le modèle markovien du système ainsi décrit. Il est toutefois à déplorer que cet algorithme soit caractérisé de la même complexité algorithmique que son homologue K tails, ce qui en restreint l'applicabilité en cas réels. L'algorithme de Synoptic a également donné lieu à l'extension Perfume [24] pour la génération de modèles de consommation de ressources. Cette extension enrichit l'apprentissage de processus markoviens supporté par Synoptic par la prise en compte d'exécutions ne différant que par leur consommation de ressources, par exemple la mémoire cache. Ce différentiel présente tout son intérêt pour la correction de systèmes dont l'exécution ne diffère pas du point de vue algorithmique mais dont la consommation de certaines ressources constitue une potentielle source d'erreurs, par exemple les équipements intelligents qui ont été analysés plus haut dans ce chapitre. Il est toutefois à déplorer que cet algorithme soit lui aussi caractérisé par la très forte complexité algorithmique qui affectait déjà K tails ainsi que Synoptic. L'algorithme de csight [18] est une autre extension de Synoptic pour l'apprentissage du comportement de systèmes concurrents. En effet de tels systèmes sont très difficiles à analyser comme il a déjà été évoqué dans la section relative aux méthodes de model checking. Pour permettre leur modélisation à partir de fichiers de logs les caractérisant csight propose une

extension de l'algorithme de Synoptic qui prend en compte leur parallélisme ainsi que leurs accès concurrents à diverses ressources ou leurs communications au travers de canaux dédiés. Cet algorithme présente un réel intérêt dans le cas où l'on aurait à caractériser le comportement de plusieurs systèmes parallèles tels que par exemple les applications exécutées par un équipement intelligent. Enfin Invarimint [17] est une dernière déclinaison de l'algorithme de Synoptic permettant de minimiser le problème de complexité algorithmique propre à cette famille de méthodes. Pour ce faire Invarimint propose une variante déclarative de l'approche supportée par Synoptic, c'est-à-dire un calcul d'automates ne comportant pas de probabilités. Bien qu'elle permette de minimiser la complexité des modèles inférés à partir d'un fichier de log cette approche souffre de l'absence de probabilités pour étiqueter ses transitions, ce qui en limite l'intérêt pour l'analyse du comportement de systèmes complexes.

L'un des principaux cas d'utilisation des méthodes d'apprentissage de processus est l'apprentissage de modèles de protocoles réseau, par exemple dans [25] où les auteurs proposent une méthode stochastique pour l'apprentissage de protocoles réseau. Cette approche propose de fusionner les états équivalents afin de construire un modèle des échanges de plusieurs systèmes logiciels. Une fois un modèle construit celui-ci peut être exploité pour calculer la probabilité d'occurence d'une succession d'évènements donnée. Les performances de cet algorithme en démontrent l'applicabilité en cas pratique. Néanmoins le manque de précision des modèles ainsi obtenus reste un obstacle à leur application pour la vérification ou la synthèse formelle de systèmes logiciels. Dans [6] les auteurs proposent une approche permettant l'apprentissage de protocoles réseau sur la base de logs de leur exécution. Cette approche se propose d'apprendre automatiquement la spécification de protocoles implémentés par des systèmes propriétaires sur la seule base de l'observation de leur comportement. Cette approche peut être intéressante dans le cadre de la validation de systèmes propriétaires dont l'implémentation n'est pas publiquement disponible tel que c'est le cas pour les applications Android. Evaluée à partir d'un ensemble de communications FTP cette approche a démontré son intérêt en exhibant une forte adéquation aux exécutions réelles. Dans [5] les auteurs proposent une approche basée sur l'interaction entre un client et un serveur échangeant des séquences de logs et des contrexemples pour la construction d'un modèle exact des protocoles réseau. Plus précisément cette approche propose que le client soumette une séquence d'évènements qui lui paraît correcte et le serveur répond si oui ou non celle-ci se trouve dans les logs d'origine. Dans le cas où la séquence proposée par le client n'est pas correcte le serveur lui indique un contrexemple qui servira à raffiner le modèle appris par le client. Au terme de ces échanges le client doit disposer d'un modèle cohérent du protocole contenu dans les fichiers de log connus par le serveur, il reste néanmoins à déplorer que comme pour beaucoup d'approches présentées dans cette section sa forte complexité algorithmique en limite l'applicabilité en cas réel. Dans [31] les auteurs proposent Discoverer, une approche permettant l'automatisation de méthodes d'apprentissage de protocoles. Cette approche se propose en particulier d'apprendre le format de requêtes attaché à un protocole donné par l'étude de traces de son exécution. Evalué à partir de deux protocoles en texte plein, du protocole HTTP ainsi que du protocole RPC Discoverer a démontré que 90% des formats qu'il avait déduit correspondait à ceux des véritables requêtes. Il reste toutefois à déplorer que l'importance des logs nécessaires à l'exécution de cet algorithme en limite l'applicabilité en cas pratiques.

Comme on l'a vu l'un des principaux obstacles à l'application de méthodes d'apprentissage de processus est l'importance des logs nécessaires à la construction de modèles exploitables. Pour solutionner ce problème plusieurs approches ont été proposées tel que par exemple le travail présenté dans [70] qui permet l'agrégation de logs basée sur des algorithmes de classification. Un travail équivalent est présenté dans [35] où les auteurs proposent de minimiser la complexité des logs donnés comme entrée aux outils d'apprentissage de processus par l'utilisation de méthodes

de classification. L'inconvénient de ces méthodes est que la classification risque d'introduire des pertes de données, c'est pourquoi l'évaluation de la qualité des modèles reste un point crucial. Dans [86] les auteurs proposent une approche permettant la validation de modèles d'apprentissage de processus sur la base de critères tels que la simplicité, l'expressivité et l'adéquation avec les logs d'entrée.

Travaux liés à l'optimisation

Finalement le dernier cas d'utilisation de méthodes formelles dans notre travail est leur application pour la résolution de problèmes d'optimisation. La problématique provient du champ des travaux en recherche opérationnelle introduits dans [60] et [53]. L'objectif de ces travaux est d'optimiser la valeur d'une certaine fonction sous un certain nombre de contraintes. Cette problématique présente souvent une forte complexité algorithmique comme dans [53] où l'on cherche à optimiser la distribution d'un produit dans plusieurs destinations en fonction de plusieurs sources, d'où la nécessité d'avoir recours à des outils issus du champ des méthodes formelles pour en faciliter la résolution.

La méthode historiquement appliquée à la résolution de ce genre de problèmes est la méthode dite du simplex introduite dans [33]. Cette méthode repose sur un modèle linéaire du problème à résoudre, autant au niveau des contraintes que de la fonction objectif. Cette contrainte de linéarité introduit une forte restriction qui limite l'expressivité de cette méthode, en particulier pour la résolution de problèmes impliquant le calcul de produits de variables. Néanmoins de nombreux travaux ont été menés pour enrichir les possibilités de modélisation des méthodes de simplex, c'est notamment le cas des travaux présentés dans [66] qui proposent une borne inférieure pour le calcul de la probabilité d'une union. Ces travaux sont théoriquement limités par la forte complexité du pire cas de l'algorithme du simplex. Néanmoins dans le cas moyen, cet algorithme reste tout à fait compétitif. Pour relaxer la contrainte de linéarité introduite par les outils de simplex solving, plusieurs méthodes de programmation non linéaire ont été proposées tels que par exemple l'utilisation de MINLP solving. Dans [65], plusieurs outils supportant les méthodes dites de MINLP solving ou de résolution de problèmes d'optimisation non linéaires sont introduits et comparés. Ces méthodes proposent essentiellement l'introduction d'opérateurs non linéaires tels que le produit ou le quotient de variables. Cependant, une trop forte utilisation de cette expressivité conduit à une dégradation de leurs performances en termes de temps de réponse. Dans [21] les auteurs introduisent νZ, une extension du SMT solver z3 pour la résolution de problèmes d'optimisation. Dans [94] les chercheurs introduisent OptiMatSAT, un outil permettant également la résolution de problèmes d'optimisation sur la base de SMT solving.

La principale critique qui peut être adressée à ces différentes méthodes est le surcoût introduit par le calcul de produits de variables qui conduit généralement à un très fort accroissement du temps de réponse et de la consommation mémorielle des algorithmes de résolution. Ces méthodes sont en effet le plus souvent caractérisées par la complexité algorithmique exponentielle inhérente aux méthodes formelles que nous avons déjà abordées dans la section précédente, ceci en rend complexe l'application à de larges problèmes impliquant de nombreuses variables et surtout de nombreuses opérations non linéaires.

2.3.5 Synthèse des travaux existants et limites

Le tableau 2.5 synthétise les limites des approches ayant trait à l'application de méthodes formelles dans notre contexte. Il existe plusieurs méthodes telles que le model checking, le SMT solving ou encore la programmation linéaire qui peuvent être appliquées tant pour la vérification

que pour la synthèse de chaînes de fonctions de sécurité, néanmoinsces approches sont caractérisées par une forte complexité algorithmique avec laquelle il est nécessaire de composer. Plusieurs travaux proposent d'ailleurs l'application de ces méthodes en vue de la validation de politiques de sécurité ou bien d'autres méthodes en vue de l'apprentissage de processus caractérisés par des logs ou encore pour la résolution de problèmes d'optimisation. La limite de ces diverses approches en reste toutefois la forte complexité algorithmique qui demande des travaux de modélisation minutieux pour permettre l'adaptation de ces méthodes à des cas pratiques requérant de hautes performances en termes de temps de réponse et de consommation mémoire.

2.4 Résumé

Dans ce chapitre, nous avons traité les travaux existants relatifs au contexte de notre étude. Le cas d'application auquel nous nous intéressons est la protection d'environnements intelligents tels que les smartphones ou tablettes sous le système d'exploitation Android. Ces environnements présentent de fortes contraintes en termes de ressources et de temps de calcul, ils sont en outre la cible de différents types d'attaques tels que les DoS, les scans de ports, les worms et les botnets. Le développement de ces différentes menaces a motivé de nombreux travaux visant à développer des solutions de protection. En outre, les faibles capacités des environnements intelligents rendent difficile le développement de méthodes de protection les concernant, leur déploiement sur les environnements risquant de dégrader l'expérience des utilisateurs.

L'une des solutions proposée pour traiter ce problème est d'avoir recours à des méthodes de virtualisation pour externaliser le déploiement de fonctions de sécurité en environnement cloud. C'est notamment le cas des solutions dites de middleboxes qui permettent l'externalisation de fonctions de sécurité, néanmoins ces approches souffrent d'une forte complexité d'administration induite sur le réseau. Pour simplifier le déploiement de ces fonctions de sécurité, il a été proposé d'avoir recours aux méthodes dites de réseaux programmables SDN. Néanmoins, la plus grande flexibilité qu'ils introduisent dans le réseau se traduit par potentiellement de nouveaux risques. Plusieurs approches ont été proposées pour adapter l'utilisation de ces méthodes au domaine de la sécurité des environnements intelligents, mais ces travaux requièrent le recours à des outils issus du champ des méthodes formelles pour en valider la cohérence.

Finalement, nous avons présenté les travaux relatifs à la vérification formelle de fonctions de sécurité en environnements programmables SDN. Le principal inconvénient de ces méthodes est la limite de leur application aux problématiques liées au plan de données et pas au plan de contrôle. Il est également à déplorer que leur complexité algorithmique en rende l'application en temps réel difficile. Pour améliorer la fidélité des modèles des systèmes à vérifier, il est possible d'avoir recours à des méthodes issues du champ de recherche autour de l'apprentissage de processus. Ces approches souffrent elles aussi d'une forte complexité algorithmique qui rend nécessaire l'utilisation de méthodes d'agrégation ou de simplification. Finalement, le dernier champ d'application de méthodes formelles intéressant notre étude est celui de la résolution de problèmes d'optimisation. Cette problématique a été explorée par plusieurs travaux de recherche, mais les méthodes de résolution présentent souvent une complexité algorithmique au-delà de la classe des problèmes polynomiaux, voire exponentiels dans certains cas particuliers.

Approches	Finalité	Complexité	Expressivité	Adaptabilité	Dynamicité
[30]	Vérification	Forte	Forte	Forte	Forte
[43]	Vérification	Forte	Forte	Forte	Forte
[97]	Vérification	Forte	Forte	Forte	Forte
[75]	Vérification	Forte	Forte	Forte	Forte
[19]	Vérification	Forte	Forte	Forte	Forte
[4]	Synthèse	Forte	Faible	Forte	Faible
[54]	Vérification, synthèse	Faible	Forte	Forte	Forte
[84]	Vérification, synthèse	Faible	Forte	Forte	Forte
[85]	Vérification, synthèse	Faible	Forte	Forte	Forte
[15]	Vérification, synthèse	Forte	Forte	Faible	Modérée
[11]	Vérification	Forte	Faible	Faible	Modérée
[46]	Vérification	Forte	Modérée	Faible	Faible
[61]	Vérification	Modérée	Modérée	Faible	Forte
[3]	Vérification	Modérée	Modérée	Faible	Modérée
[59]	Vérification	Forte	Faible	Faible	Modérée
[40]	Vérification	Forte	Forte	Faible	Modérée
[2]	Vérification	Forte	Forte	Faible	Faible
[24]	Vérification	Modérée	Forte	Faible	Modérée
[20]	Apprentissage	Forte	Forte	Forte	Faible
[16]	Apprentissage	Forte	Forte	Forte	Faible
[80]	Apprentissage	Forte	Forte	Forte	Faible
[18]	Apprentissage	Forte	Forte	Forte	Faible
[17]	Apprentissage	Modérée	Modérée	Forte	Faible
[25]	Apprentissage	Faible	Faible	Faible	Forte
[6]	Apprentissage	Forte	Modérée	Modérée	Faible
[5]	Apprentissage	Forte	Modérée	Faible	Modérée
[31]	Apprentissage	Forte	Modérée	Modérée	Faible
[70]	Apprentissage	Forte	Forte	Forte	Faible
[35]	Apprentissage	Forte	Forte	Forte	Faible
[33]	Optimisation	Modérée	Modérée	Forte	Forte
[65]	Optimisation	Forte	Forte	Forte	Forte
[21]	Optimisation	Forte	Forte	Forte	Forte
[94]	Optimisation	Forte	Forte	Forte	Forte

FIGURE 2.5 – Tableau de synthèse des travaux sur les méthodes formelles

3

Orchestration pour la protection d'applications Android

Le premier objectif de notre travail dans cette thèse était de définir l'architecture d'un orchestrateur responsable de la gestion des chaînes de fonctions de sécurité. En particulier, un tel orchestrateur doit pouvoir répondre aux besoins en termes de sécurité des utilisateurs du réseau, tout en équilibrant le coût du déploiement des chaînes. Pour ce faire, le processus d'orchestration de chaînes s'appuie sur différentes méthodes telles que l'apprentissage de processus, la synthèse et la vérification formelles de systèmes logiciels, des algorithmes de regroupement et enfin des techniques d'optimisation linéaire, voire non-linéaire pour en minimiser le coût de déploiement. En particulier parmi toutes ces méthodes, l'apprentissage de processus occupe une position toute particulière puisque c'est lui qui initie et oriente tout le reste du processus d'orchestration à proprement parler, nous avons donc retenu de le présenter dans ce chapitre. Nous commencerons par présenter l'insertion de l'orchestrateur dans le réseau, avant d'en décrire l'architecture interne. Nous discuterons ensuite notre processus d'apprentissage du comportement d'applications Android qui joue un rôle central dans notre stratégie d'orchestration. En premier lieu, nous détaillerons la stratégie de collecte des flux réseau d'une application sur la base d'un agent dédié déployé directement sur les équipements intelligents. En second lieu, nous présenterons notre stratégie d'agrégation des flux réseau d'une application en vue d'en minimiser l'hétérogénéité. En troisième lieu, nous détaillerons notre algorithme de construction du modèle markovien d'une application à partir des flux agrégés et finalement nous conclurons ce chapitre en présentant l'implémentation de notre prototype et les résultats expérimentaux que nous avons obtenus.

3.1 Architecture réseau de notre approche d'orchestration

3.1.1 Insertion de l'orchestrateur dans le réseau

L'architecture dans laquelle s'intègre notre orchestrateur est décrite sur la figure 3.1. Celle-ci présente trois entités particulières : (i) sur la gauche, l'équipement intelligent avec plusieurs applications nécessitant une protection avec un agent intégré (qui peut typiquement être instancié par un switch openflow virtuel pour des problématiques de redirection) ; (ii) au milieu, l'infrastructure d'un fournisseur de cloud fournissant plusieurs fonctions de sécurité ainsi qu'un gestionnaire de sécurité ; (iii) sur la droite, les destinations distantes interagissant avec l'application exécutée par l'équipement intelligent. Le gestionnaire de sécurité se compose de différentes

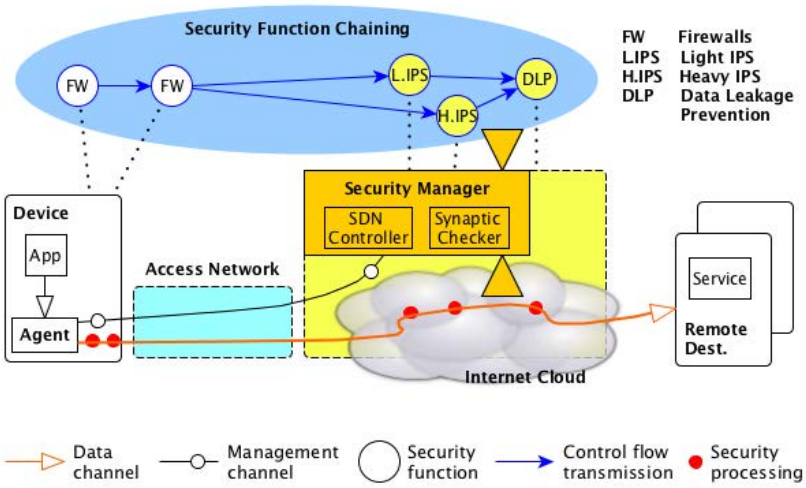

FIGURE 3.1 – Intégration de notre orchestrateur au sein de l'architecture d'un réseau SDN

entités : premièrement le contrôleur du réseau SDN, ensuite notre orchestrateur en charge de la gestion des chaînes de fonctions de sécurité, utilisant un ou plusieurs langages de haut niveau tel que par exemple Frenetic ou Pyretic. En plus de l'extension Kinetic de ces langages qui proposent des dispositifs de vérification pour le plan de contrôle (en bleu), il utilise notre travail pour valider la correction du plan de données (chemin entre l'agent et le service, en rouge). Pour ce faire, notre orchestrateur s'appuie sur une conjonction de méthodes de synthèse formelle basée sur des règles d'inférence logique qui seront décrites au chapitre 4, ainsi que de vérification formelle par application de model checking ou de SMT solving sur un modèle abstrait des chaînes de sécurité tel qu'il sera présenté au chapitre 5.

Finalement, les fonctions de sécurité peuvent être déployées soit sur l'équipement intelligent, soit dans une infrastructure cloud. L'architecture d'orchestration que nous proposons s'inscrit dans le contexte de la protection d'environnements intelligents, tels que des smartphones ou des tablettes. Toutefois, notre travail n'est pas limité à ce cas d'utilisation particulier, en effet les chaînes de fonctions de sécurité sont utilisées pour d'autres équipements requérant une sécurité très flexible et adaptable tels que par exemple les réseaux de capteurs ou d'objets connectés. Lorsqu'une application initie une communication avec une destination distante, tous les messages émis et reçus par cette application passent par l'agent (le switch virtuel) de l'équipement intelligent. Le switch contactera alors le contrôleur SDN du fournisseur réseau afin de savoir comment rediriger les messages pour effectuer les contrôles de sécurité. En fonction des risques et du contexte, le gestionnaire de sécurité active les fonctions de sécurité correspondantes.

3.1.2 Relation entre l'orchestrateur et les équipements intelligents

La chaîne de traitements est donc initiée par les équipements intelligents qui doivent faire remonter leur besoin de protection auprès de l'orchestrateur. Pour ce faire les équipements intelligents doivent premièrement s'inscrire auprès de l'orchestrateur, par exemple en téléchargeant un client dédié. Une fois connu de l'orchestrateur les équipements intelligents peuvent faire remonter leurs besoins en termes de protection, par exemple en notifiant la liste de leurs applications connectées à internet. Cette connaissance sur les applications ayant besoin de protection introduit au demeurant de nouvelles problématiques de confidentialité, il serait par exemple né-

cessaire de songer à une anonymisation des listes d'applications afin de préserver la vie privée des utilisateurs, par exemple en transmettant une liste de noms d'applications communes à plusieurs équipements ayant fait la requête d'une chaîne à notre orchestrateur.

Notre spécification des besoins de sécurité propres à une application mobile repose sur deux sources différentes : les permissions demandées par l'application ; le descriptif de ses interactions réseau. Les permissions seront utilisées par notre orchestrateur afin de disposer d'un modèle des données manipulées par l'application en vue de prévenir d'éventuelles fuites de données sensibles vers des destinations non sécurisées. La liste des permissions relatives à une application peut être obtenue par deux moyens différents, soit en téléchargeant et en extrayant le fichier de manifest de l'application depuis Google Play Store soit en les collectant au travers de l'interface Java mise à disposition par le système Android sur les équipements intelligents. Le descriptif des interactions réseau d'une application sera quant à lui utilisé pour repérer d'éventuelles attaques ou des connexions avec des destinations douteuses. Nous appuyons notre modèle d'interactions sur la notion de flux réseau définie précédemment, ce modèle ne permet pas d'avoir accès au contenu des messages générés par l'application. Ce manque peut être compensé par l'utilisation des permissions qui permettent d'approcher un modèle des données de l'application. Nous collectons les flux générés par une application au travers d'un agent dédié [68] déployé directement sur l'équipement intelligent. La collecte des flux est cependant un procédé long et coûteux en ressources de calcul du côté de l'équipement intelligent. Pour économiser cette opération nous procédons à la collecte des flux seulement en cas de première découverte d'une application ou de mise à jour majeure de son fonctionnement, puis nous stockons les flux en base de données. Lorsque la même application sera redécouverte ultérieurement, ces flux pourront alors être réutilisés pour générer la chaîne correspondante sans avoir à procéder une nouvelle fois à la collecte au niveau de l'équipement intelligent. Pour permettre une gestion optimale de notre orchestrateur, il pourrait être intéressant d'envisager un apprentissage fait en amont du comportement de plusieurs applications sélectionnées par exemple sur la base de critères de popularité au niveau des marchés.

3.1.3 Relation entre l'orchestrateur et les chaînes de fonctions de sécurité

La composition de fonctions de sécurité est réalisée par l'orchestrateur sur la base de plusieurs facteurs, incluant les applications à protéger, les destinations qu'elles contactent et enfin les propriétés du réseau. Pour donner quelques exemples, les fonctions de sécurité peuvent inclure des pare-feux de niveau applicatif ou bien réseau, des systèmes de détection d'intrusion et des mécanismes de prévention de fuite de données. Ces fonctions ne sont pas limitées à l'analyse du traffic, elles peuvent aussi inclure des fonctions analysant la configuration des équipements intelligents et de leurs applications. Enfin ces chaînes peuvent être soit construites, soit sélectionnées et vérifiées sur la base de plusieurs critères, l'importance d'assurer de hautes performances édictant le choix de l'une ou l'autre méthode.

La construction des chaînes de fonctions de sécurité se fait à partir des propriétés de sécurité devant être garanties pour protéger les applications clientes de notre orchestrateur. Au terme de ce processus les chaînes sont représentées sous la forme de spécifications fonctionnelles à un haut niveau d'abstraction dans un langage de programmation dédié, par exemple Pyretic ou Frenetic. Ces spécifications sont ensuite compilées en règles de configuration OpenFlow ou bien sous la forme de règles de configuration de fonctions de sécurité virtualisées, l'utilisation de langages de programmation de haut niveau permet de découpler la construction des chaînes de leur implémentation concrète dans le réseau.

L'orchestrateur est tout particulièrement en charge d'assurer la correction des chaînes de

fonctions de sécurité déployées dans le réseau. Comme nous avons déjà eu plusieurs fois l'occasion de l'évoquer, la composition de chaînes de fonctions de sécurité est un processus complexe sujet à de nombreuses erreurs dont le déploiement aurait de funestes conséquences sur la protection des applications clientes de notre orchestrateur. Pour prévenir ces risques notre orchestrateur s'appuie sur plusieurs outils issus du champ des méthodes formelles tels que les méthodes de vérification algorithmique (model checking), de résolution de contraintes SMT et de programmation logique. D'autres méthodes permettent également l'optimisation du déploiement des chaînes de fonctions de sécurité en vue de minimiser l'impact de leur déploiement sur le réseau, néanmoins la gestion de leur haute complexité algorithmique fera l'objet de plusieurs discussions et évaluations de performances dans les chapitres 4 et 5 de ce mémoire.

Finalement, une fois que les chaînes ont été construites, validées, optimisées et déployées l'orchestrateur reste en charge de leur reconfiguration. Cette reconfiguration peut advenir suite à différents évènements réseau, par exemple la connexion d'une nouvelle station mobile à protéger, une mise à jour majeure d'une des applications à protéger ou encore l'indisponibilité d'un des équipements supportant la chaîne dans le réseau. Pour chacun de ces cas de figure notre orchestrateur doit assurer le calcul d'un nouveau plan de déploiement de ses chaînes de fonctions de sécurité ou de leurs règles, ces aspects seront détaillés plus longuement dans le chapitre suivant de ce mémoire.

3.1.4 Relation entre l'orchestrateur et le réseau SDN

Pour permettre le déploiement des chaînes de fonctions de sécurité, notre orchestrateur s'appuie sur une infrastructure SDN qui peut être mise à disposition par un fournisseur cloud. Une fois, le programme d'une chaîne composé à un haut niveau d'abstraction, celui-ci est compilé en règles OpenFlow ou en règles de fonctions virtuelles et transmis au contrôleur du réseau. En déployant ces règles dans le réseau du fournisseur cloud, le contrôleur compose les fonctions de sécurité, afin de construire la chaîne appropriée et notifie les switches. Les règles dirigent les messages entrants au sein de la chaîne avant de les transmettre à leur destination finale. Les fonctions de sécurité peuvent être hébergées localement sur l'équipement intelligent ou dans le cloud mis à disposition par le fournisseur.

Pour permettre l'optimisation du déploiement des chaînes de fonctions de sécurité, notre orchestrateur a besoin de connaître certaines informations sur le réseau, notamment sa topologie et la capacité de ses différents composants. Cette information est accessible au travers de la machine virtuelle associée aux langages de programmation de contrôleur que nous utilisons. Ceux-ci s'appuient sur les fonctionnalités mises à disposition par le contrôleur du réseau, afin de collecter les informations à transmettre à notre orchestrateur. Finalement, des informations peuvent également être transmises en temps réel par le réseau à notre orchestrateur au travers de mécanismes d'alerte qui seront détaillés dans le chapitre suivant.

Il est également possible d'envisager le problème sous le point de vue inverse, à savoir que c'est la spécification de la chaîne qui va être le point de départ de l'ajustement des ressources du réseau. En effet, en contexte SDN, il est possible de reprogrammer le réseau en temps réel à la demande des utilisateurs, la spécification d'une chaîne de fonctions de sécurité serait alors utilisée pour faire du provisioning. Dans ce contexte, les besoins seraient transmis au contrôleur du réseau qui se chargerait alors de mettre en place les ressources nécessaires au travers d'un plan de déploiement dédié. Dans le cadre d'une approche d'optimisation ou de provisioning, il est de toute façon nécessaire d'avoir recours à des méthodes dédiées afin de construire le meilleur plan de déploiement des chaînes de fonctions de sécurité au sein du réseau tout en assurant de hautes performances en termes de temps de réponse.

3.2 Architecture logicielle de notre orchestrateur

Après avoir décrit l'insertion de notre orchestrateur dans son contexte et les interactions qui les lient, il convient désormais d'en présenter l'architecture interne. Celle-ci est présentée sur le graphique 3.2 qui en détaille les principaux composants et les interactions qui les lient.

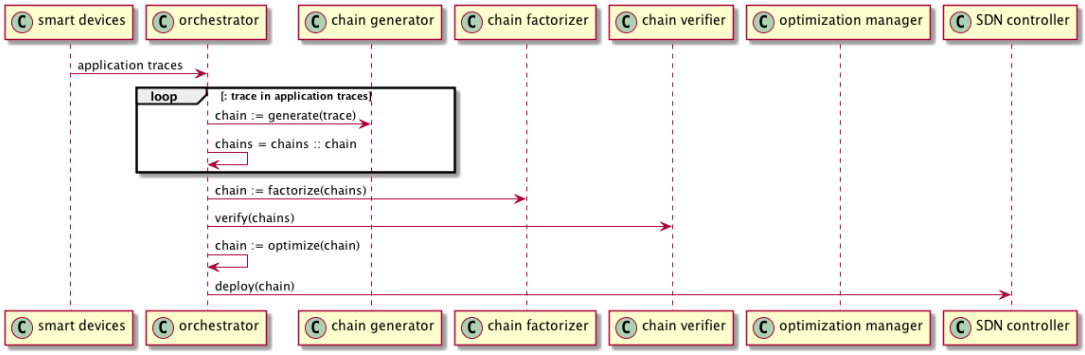

FIGURE 3.2 – Architecture de notre orchestrateur de chaînes de fonctions de sécurité

Ce graphique présente différents modules interagissant les uns avec les autres. En premier lieu, on reconnaît les équipements intelligents et le contrôleur SDN dont il a déjà été question dans la précédente section. Les traces représentant le comportement réseau des applications à protéger sont donc transmises à notre orchestrateur qui va les traiter itérativement afin de construire les chaînes correspondantes. Une fois construites ou sélectionnées, ces chaînes sont ensuite factorisées afin de construire une seule large chaîne à déployer dans le réseau. Cette chaîne est ensuite vérifiée par application de model checking ou de SMT solving pour s'assurer du fait qu'elle ne contient pas de règles contradictoires. La dernière étape de notre processus d'orchestration consiste à optimiser le plan de déploiement de cette large chaîne de fonction de sécurité en fonction de paramètres contextuels.

Il est en outre nécessaire de préciser que la construction des chaînes de fonctions de sécurité fait elle-même appel à plusieurs étapes décrites dans le diagramme 3.3. Le premier de ces modules est responsable d'apprendre le comportement réseau d'une application à protéger sous la forme d'un automate markovien. Cet automate ainsi que les permissions déclarées par l'application correspondante sont ensuite transmis au module d'apprentissage de propriétés qui va en extraire les propriétés logiques correspondantes. Finalement, ces propriétés sont transmises au module de synthèse qui va générer une chaîne correspondante sur la base de règles d'inférence dédiées spécifiées à l'aide de méthodes de logic programming.

Le goulot d'étranglement de cette architecture est le module d'apprentissage de comportement : il peut prendre plusieurs minutes pour élaborer l'automate associé avec une application. Après avoir été générées les chaînes de fonctions de sécurité sont enregistrées dans une base de données dédiée, où elles sont indexées sur la base du nom de l'application pour laquelle elles ont été générées. Ainsi l'apprentissage et la génération sont découplés de la factorisation, de l'optimisation et du déploiement et les chaînes de fonctions de sécurité peuvent être chargées efficacement en temps réel.

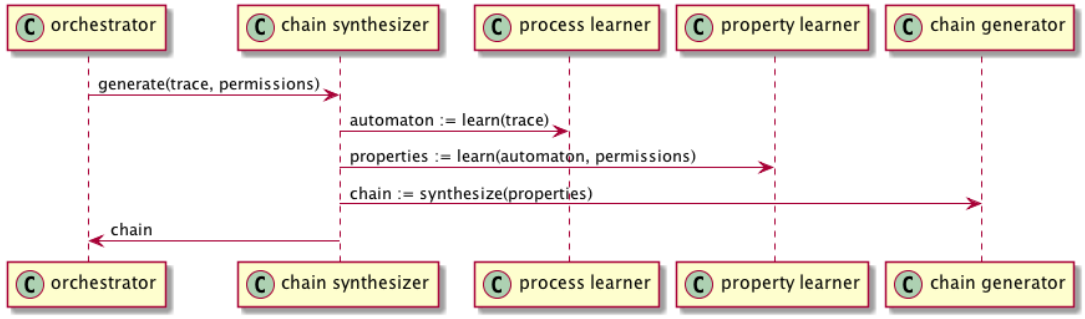

FIGURE 3.3 – Architecture interne du module de synthèse de chaînes de fonctions de sécurité

3.3 Apprentissage du comportement réseau d'applications

Nous décrivons ci-dessous les différentes étapes relatives à l'apprentissage du comportement réseau d'applications Android.

3.3.1 Collecte des données relatives aux flux réseau

Comme indiqué plus haut, nous collectons les flux relatifs à une application via un agent déployé directement sur les équipements intelligents. Cet agent, Flowoid [68] a été développé dans le cadre d'un travail relatif à cette thèse. Il propose la collecte de différentes informations relatives aux flux réseau d'une application et en particulier :

— un timestamp indiquant le début de la communication ;
— le nom du paquet ayant produit le flux ;
— l'adresse IP de l'équipement intelligent ;
— le numéro de port utilisé par l'équipement intelligent ;
— l'adresse IP de la destination ;
— le numéro de port utilisé par la destination ;
— le protocole réseau utilisé pour le flux (TCP ou UDP) ;
— le nombre de paquets transmis dans le flux ;
— le nombre d'octets transmis dans le flux.

Flowoid permet également la collecte d'autres informations qui ne seront pas exploitées ultérieurement, nous ne les présenterons donc pas ici. La collecte de ces flux est réalisée au travers du déploiement de Flowoid directement sur l'équipement intelligent : lorsqu'une application est active et connectée à internet celle-ci communique avec des destinations distantes. Les flux générés dans le cadre de ces communications sont alors collectés par Flowoid et enregistrés localement dans l'attente d'être retransmis à notre orchestrateur ou à une plateforme chargée de les enregistrer dans une base de données dédiées. Quoi qu'il en soit cette étape de collecte prend un temps relativement long du fait de la nécessité de placer l'application en contexte réaliste, il pourrait être envisagé de l'automatiser au travers de déploiements en sandbox mais ceci outrepasse le cadre de notre travail. Une fois qu'ils ont été collectés sur l'équipement intelligent les flux relatifs à une application doivent être centralisés. Cette opération se fait au moyen d'un protocole dédié, typiquement le protocole NetFlow [28].

Un dataset privé comportant les traces de plusieurs applications Android a été constitué dans notre laboratoire hors du cadre de cette thèse. Ce dataset comporte en outre les fichiers manifests de plusieurs des applications dont les flux ont été enregistrés, la présence d'un fichier de manifest n'est pas garantie pour chaque application. Nos expérimentations dans le cadre de cette thèse se

Applications	Nombre de lignes	Adresses IP	Manifest	Permissions
disneyland	282	5	non	-
dropbox	1000	17	oui	5
faceswitch	151	30	oui	3
lequipe	1000	151	non	-
meteo	1000	80	non	-
ninegag	1000	88	non	-
pokemongo	275	24	oui	6
ratp	779	3	non	-
skype	1000	161	oui	11
viber	1000	78	oui	15

FIGURE 3.4 – Descriptif des applications considérées dans le cadre de nos travaux

sont principalement concentrées sur un ensemble de dix applications Android particulières dont le descriptif est fourni dans le tableau 3.4. Pour chaque application nous indiquons le nombre de flux contenus dans l'enregistrement, le nombre d'adresses IP contactées au travers de ces flux, la présence éventuelle d'un fichier de manifest et le cas échéant le nombre de permissions dangereuses enregistrées.

3.3.2 Stratégies d'agrégation des flux réseau

Comme détaillé dans la section correspondante des travaux relatifs la principale limite des méthodes d'apprentissage de processus réside dans leur difficulté à traiter des enregistrements très disparates. Dans notre contexte où nous considérons des paires d'adresse IP et de numéro de port comme décrivant un flux il devient donc nécessaire d'avoir recours à des stratégies d'agrégation pour limiter la taille des entrées fournies aux méthodes d'apprentissage de modèle. Nous présentons ici deux stratégies d'agrégation que nous avons utilisées dans notre travail, l'une sur les adresses IP et l'autre sur les numéros de ports.

La première opération que nous exécutons dans le cadre de notre procédure d'apprentissage du comportement d'une application consiste à enrichir le modèle des flux collectés par Flowoid avec l'information relative à l'organisation possédant l'adresse IP contactée pour un flux. Cette information, abrégée en orgname dans la suite de ce mémoire, peut être collectée au moyen de la commande shell whois qui retourne une chaîne de caractères décrivant les informations relatives à l'adresse IP qui lui a été passée en paramètre. En plus de l'information relative à l'orgname whois peut encore retourner une seconde information relative au nom du réseau dans lequel l'adresse IP a été déployée, cette information est appelée netname dans la suite de ce mémoire.

Les champs orgname et netname peuvent être tous deux utilisés dans le cadre de notre procédure d'apprentissage avec la distinction que le champ netname est plus distinctif que le champ orgname. Pour identifier celui des deux champs devant être ajouté aux flux d'une application nous comptons le nombre d'occurences de chaque netname dans la trace d'une application : si ce nombre dépasse un certain seuil donné en paramètre à notre procédure d'apprentissage alors ce sera le champ netname qui sera ajouté, dans le cas contraire ce sera le champ orgname. Cette procédure permet l'agrégation des flux d'une application en un ensemble restreint de destinations à considérer : cette minimisation de la taille de l'entrée à considérer pour chaque application est en effet l'un des éléments cruciaux permettant l'application des méthodes que nous décrirons plus loin à la problématique de l'orchestration de chaînes de fonctions de sécurité.

Du fait de l'utilisation de requêtes whois cette phase d'agrégation peut prendre un certain temps (jusque plusieurs minutes) et ne peut donc pas être effectuée en temps réel. Comme

indiqué plus haut nous enregistrons le résultat de cette première étape de notre processus d'apprentissage en base de données afin de pouvoir le réutiliser ultérieurement. Au niveau de notre procédure d'apprentissage nous calculons également d'autres métriques relatives aux adresses IP contactées par une application et notamment leur nombre d'occurences, une mesure de leur ranking BGP Border Gateway Protocol qui sera utilisée ultérieurement et enfin un flag indiquant le status qui leur est associé, nous aurons l'occasion d'y revenir.

Nous complétons cette stratégie pour l'agrégation d'adresses IP par une seconde pour l'agrégation de numéros de ports : bien que la plupart des applications Android n'utilisent que les ports HTTP et HTTPS et n'ont donc pas besoin d'une telle stratégie d'agrégation, certaines applications présentent une forte disparité de numéros de ports qui requièrent donc d'être agrégés. Dans ce cas, nous pouvons utiliser les préfixes de numéros de ports les plus fréquents afin de agréger les flux : cette approche nous permet d'agréger ensemble des flux qui ont le même numéro de port ; de plus quand une application communique fréquemment sur une certaine gamme de ports nous pouvons agréger ces flux afin de obtenir une représentation plus compacte du traffic de l'application. L'idée est de nouveau de ne considérer que les préfixes apparaissant plus souvent qu'un certain seuil afin de limiter l'influence de numéros de ports n'apparaissant qu'une ou deux fois dans les traces considérées.

3.3.3 Algorithme de construction du modèle de comportement réseau d'une application Android

Après avoir collecté et enrichi la trace des flux d'une application nous les utilisons pour construire son modèle de comportement. Conformément aux informations présentées dans le contexte et les travaux relatifs nous proposons de construire un modèle des communications d'une application, d'où notre choix de nous orienter vers les méthodes d'apprentissage d'automates. Nous avons eu un intérêt tout particulier pour les méthodes d'apprentissage d'automates markoviens tel que l'algorithme de K tail [20] ou encore ses extensions Synoptic [16] dont Invarimint [17] est une version dite déclarative. Ces deux approches présentent certaines limites dans le cadre de notre travail : comme indiqué dans les travaux relatifs Synoptic tend à produire des automates très complexes tandis qu'Invarimint produit des automates plus simples mais dépourvus de probabilités. Pour donner un exemple, nous considérons les automates générés pour l'une des applications de notre dataset avec chacune de ces méthodes, nous commençons par Synoptic dont l'automate résultant est présenté en figure 3.5. Les états de cet automate sont calculés en considérant les paires orgname / numéro de port contactés pour chaque destination. Comme on peut le voir Synoptic conduit à produire un automate complexe contenant beaucoup de redondances et limitant donc la signification des probabilités associées avec les transitions correspondantes. L'application d'Invarimint sur le même fichier de logs conduit à la production d'un automate certes plus simple mais cette fois-ci totalement dépourvu de probabilités comme présenté dans le graphique 3.6.

En synthèse, nous avons cherché à construire un modèle d'applications qui réunisse le meilleur de ces deux approches, à savoir la simplicité garantie par Invarimint et le calcul des probabilités assuré par Synoptic. Cet automate est implémenté par deux tables : *Etats* associe les flux à leur nombre d'instances, tandis que *Transitions* associe des paires d'états à la probabilité de succession correspondante. Les arguments de notre algorithme sont les listes notées *App* de flux étendus calculés par notre algorithme de regroupement pour une application et la taille N de cette liste ; les variables locales que nous utilisons sont *flux* contenant l'orgname associé avec un flux et *transition* qui contient deux flux consécutifs. Les étapes de l'algorithme d'apprentissage 1 sont les suivantes : chaque flux réseau est assigné à un état de l'automate en fonction de l'orgname qui lui est associé, on comptabilise ainsi le nombre de flux par état. En fonction de la succession des états dans la liste des flux on identifie les transitions de l'automate, celles-ci sont affectées d'un poids analogue à celui des états. Une fois parcourue la liste de tous les flux l'algorithme se termine par le calcul des probabilités associées à chaque transition, celles-ci sont le résultat du ratio entre le nombre d'occurences d'une transition et le nombre d'occurences de son état d'origine. Les états de l'automate peuvent être enrichis par de l'information additionnelle afin de représenter plus de détails sur le traffic correspondant. Concrètement, nous calculons les métriques réseau standard suivantes pour chaque état de l'automate, celles-ci seront utilisées dans le chapitre 4 pour la génération des chaînes de fonctions de sécurité :

- les nombres d'octets et de paquets maximums transmis pendant une communication ;
- le nombre moyen de paquets et d'octets transmis pendant une communication ;
- la durée du flux le plus long ;
- la durée moyenne de l'ensemble de flux ;
- le ratio entre traffic TCP et traffic UDP.

3.3.4 Exemples d'automates

Pour illustrer le gain apporté par notre approche en termes d'apprentissage d'automates, nous reprenons le même exemple que précédemment dans la figure 3.7. Nous avons cherché à obtenir un modèle présentant le moins

Algorithme 1 Algorithme d'apprentissage de l'automate

```
Etats := ∅
Transitions := ∅
App :=Liste des flux.
                                          ▷ Initialise l'ensemble des états
flux := App[0]
Etats[flux] := 1
                                          ▷ Compte les occurences d'états et de transitions
for i ∈ 1..N do
    transition := (flux, App[i])
    flux := App[i]
    if flux ∈ Etats then
        Etats[flux] += 1
    else
        Etats[flux] := 1
    end if
    if transition ∈ Transitions then
        Transitions[transition] += 1
    else
        Transitions[transition] := 1
    end if
end for
                                          ▷ Calcule la probabilité de chaque transition
for transition ∈ Transitions do
    Transitions[transition] :=
    Transitions[transition]/Etats[transition_0]
end for
```

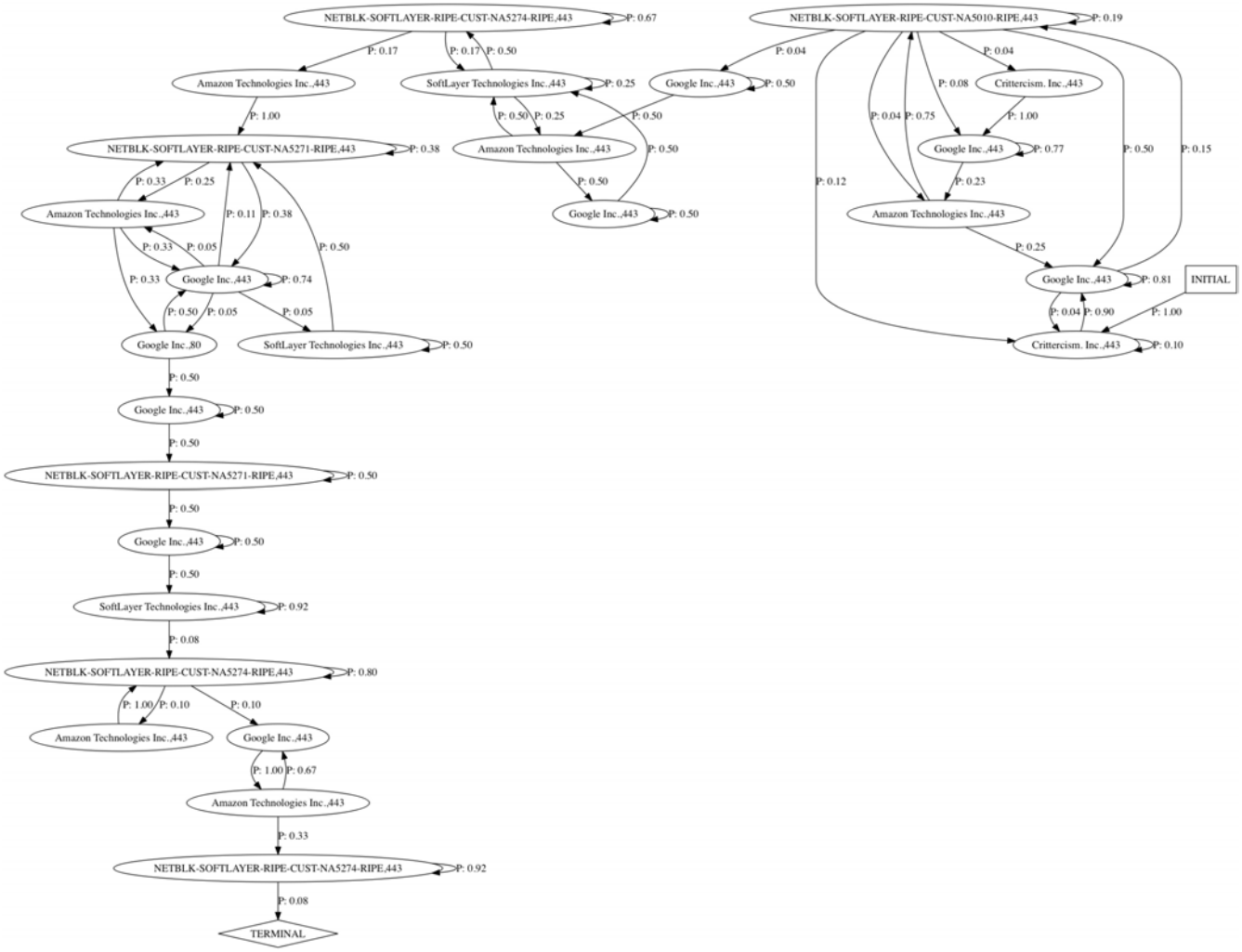

FIGURE 3.5 – Automate obtenu par Synoptic pour l'application considérée

d'états possibles pour en faciliter l'exploitation en vue de la détection d'attaques, en effet la redondance introduite par Synoptic rendait plus ardue une telle exploitation. De même, nous attendions un automate caractérisé par des probabilités afin de pouvoir mieux identifier les déviations d'un comportement standard, c'est la raison pour laquelle nous avons cherché à nous démarquer d'Invarimint.

3.4 Prototype et résultats expérimentaux

Nous avons évalué les performances de notre algorithme d'apprentissage au travers de plusieurs expérimentations. En particulier nous voulions comparer les performances obtenues avec différentes méthodes d'apprentissage d'automate. Le contexte expérimental était basé sur un ordinateur portable MacBook Air avec un processeur Intel Core i5 (1.7 GHz) et 4Gb de RAM. Nous avons considéré les trois méthodes d'apprentissage de processus suivantes : Synoptic, sa version déclarative implémentée dans Invarimint, et finalement notre propre méthode d'apprentissage décrite dans la section précédente. Pour ces expérimentations nous avons considéré l'ensemble de dix applications décrites dans la figure 3.4. A partir de ces fichiers de log nous construisons les automates avec les trois approches que nous avons retenues, à signaler que nous n'avons pas pu obtenir les automates pour les applications lequipe, skype and viber avec Synoptic à cause d'une consommation mémoire excessive. Afin comparer les automates que nous obtenons nous avons défini les critères d'évaluation suivants :

— la simplicité de l'automate résultant, exprimée en termes de nombre d'états et de transitions ;
— la précision du modèle, exprimée en termes de nombre de flux d'autres applications rejetées par un

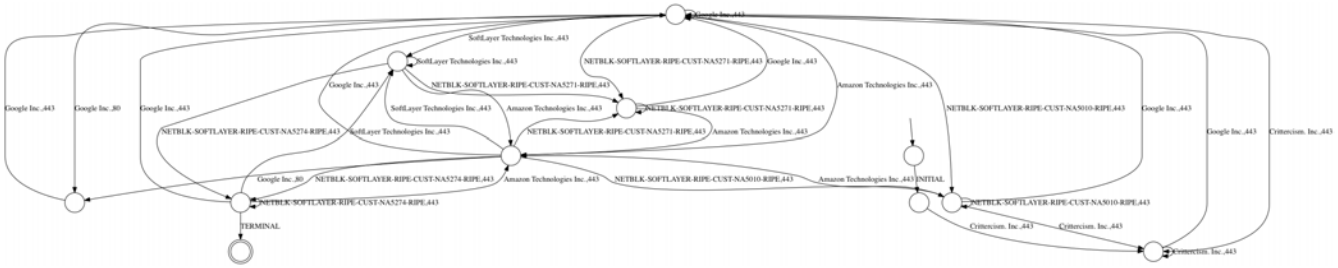

FIGURE 3.6 – Automate obtenu par Invarimint pour l'application considérée

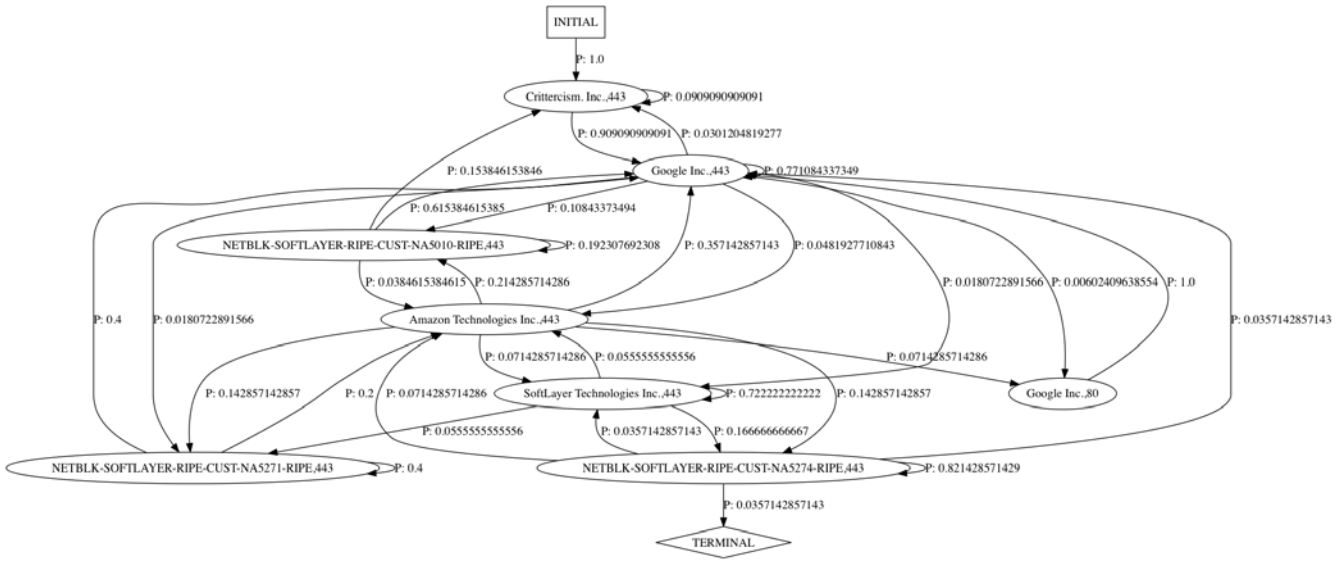

FIGURE 3.7 – Automate obtenu par notre approche pour l'application considérée

automate.

3.4.1 Prototypage

Le diagramme de classes décrivant le prototype de notre orchestrateur est décrit dans la figure 3.8. Il comprend un total de 13405 lignes de code Python dont 1741 pour les seuls modules présentés sur le diagramme 3.8, les 80% restant correspondant à des modules intervenant en prolongement des gestionnaires présentés ici.

Ce diagramme présente au même niveau les différents composants de notre orchestrateur, les trois sous-composants du bloc intitulé "chain synthesizer" du diagramme 3.2 sont présentés à la suite les uns des autres pour éviter de surajouter des classes n'ayant pour seul objectif que d'offrir une façade de programmation.

Les travaux relatifs à l'apprentissage du comportement réseau d'applications Android présentés dans ce chapitre correspondent au module ProcessLearner du graphique 3.8. Cette classe représente un total de 417 lignes de code. Ce module présente les méthodes nécessaires pour charger la liste des logs relative à une application ainsi que pour effectuer les opérations d'agrégation et d'apprentissage présentées dans ce chapitre. Des méthodes analogues sont présentées pour construire un modèle du comportement d'une application au niveau système à partir d'enregistrements du comportement des applications à ce niveau, toutefois nous n'avons pas exploité cet aspect dans le cadre de nos travaux en raison de la difficulté à exploiter les enregistrements correspondants dans une approche de profilage.

3.4.2 Evaluation de la simplicité des automates

Nous avons uniquement considéré des méthodes exactes de génération dans ces évaluations : les automates produits par de tels algorithmes accepteront tous les logs utilisés pendant l'apprentissage. Toutefois de tels auto-

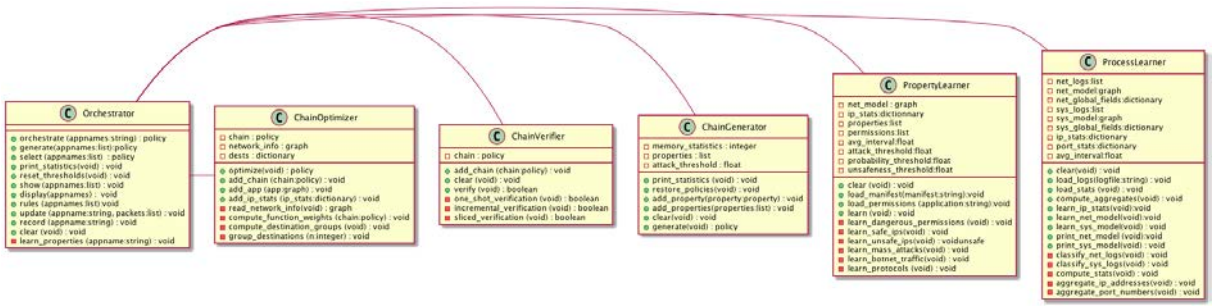

FIGURE 3.8 – Diagramme de classes décrivant le prototype de notre orchestrateur

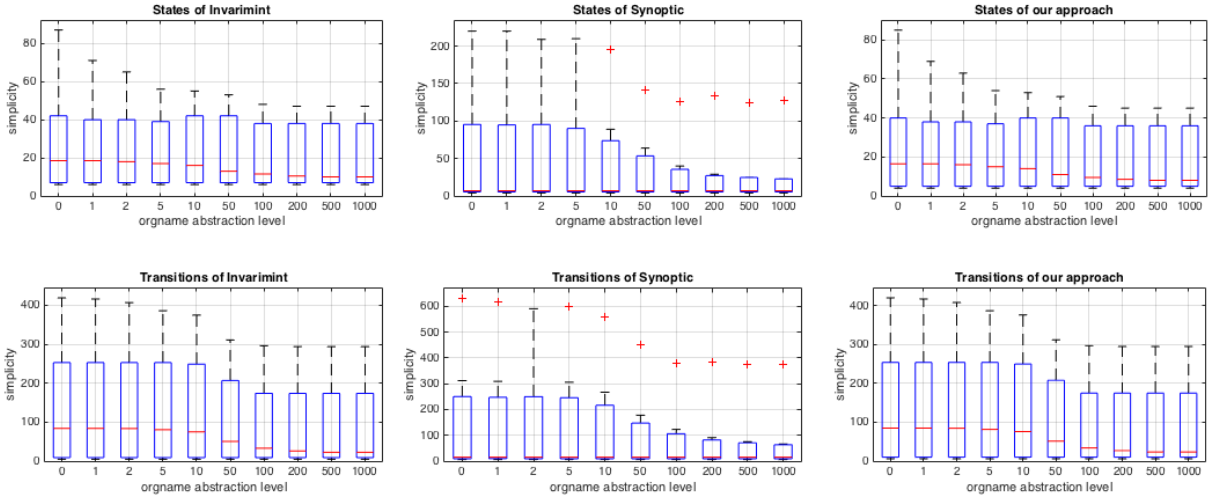

FIGURE 3.9 – Simplicité des automates générés avec les méthodes considérées

mates peuvent être très compliqués à lire et à interpréter, et leur taille a une influence sur le nombre de règles SDN à générer. Nous avons donc choisi leur simplicité comme un critère d'évaluation, mesurée en termes de nombre d'états et de transitions d'un automate. Parce que cette métrique varie tout à la fois avec la méthode de génération et le niveau d'abstraction des logs d'entrée, nous avons comparé les différentes méthodes en les appliquant sur des logs agrégées en utilisant différentes valeurs du paramètre de seuil n qui correspond à la valeur minimale à partir de laquelle une adresse IP est considérée comme étant significative. Les résultats de ces expérimentations sont présentés dans la figure 3.9.

Sans agréger les adresses IP notre approche produit des automates avec en moyenne 105.2 états et 322.8 transitions pour ce dataset. Les automates produits par Invarimint et par notre approche sont de complexité similaire en moyenne, Invarimint produit des automates qui ont 2 états de plus et une transition de moins que notre approche. Au contraire Synoptic produit des automates avec en moyenne 11.2 états et 29 transitions de plus que notre approche, il est en outre nécessaire de préciser que ces résultats sont biaisés du fait que Synoptic n'a pas pu être exécuté avec succès pour skype, viber et lequipe du fait d'une consommation mémoire excessive. De fait, les automates générés par Synoptic sont toujours les plus complexes. Au point de vue des performances de la stratégie d'agrégation, la meilleure amélioration est observée en remplaçant les adresses IP par les orgnames correspondants ; concernant l'influence de la valeur de n, le plus fort impact est observé lorsqu'on agrège les netnames les moins fréquents, signifiant que limiter l'influence des valeurs avec une basse fréquence d'apparition est le facteur le plus important pour choisir la valeur de seuil n. Pour compléter ces résultats nous avons aussi comparé la taille des automates pour notre approche et pour Invarimint lorsque l'on applique la stratégie d'abstraction des numéros de ports sur les logs des trois applications les plus complexes : lequipe, skype et viber, Synoptic ayant de fait été écarté de par son impossibilité de générer un automate pour ces applications. Durant ces expériences nous avons considéré l'influence du seuil de sur les numéros de ports variant de 0 à 10. Ces résultats sont décrits dans la Figure 3.10.

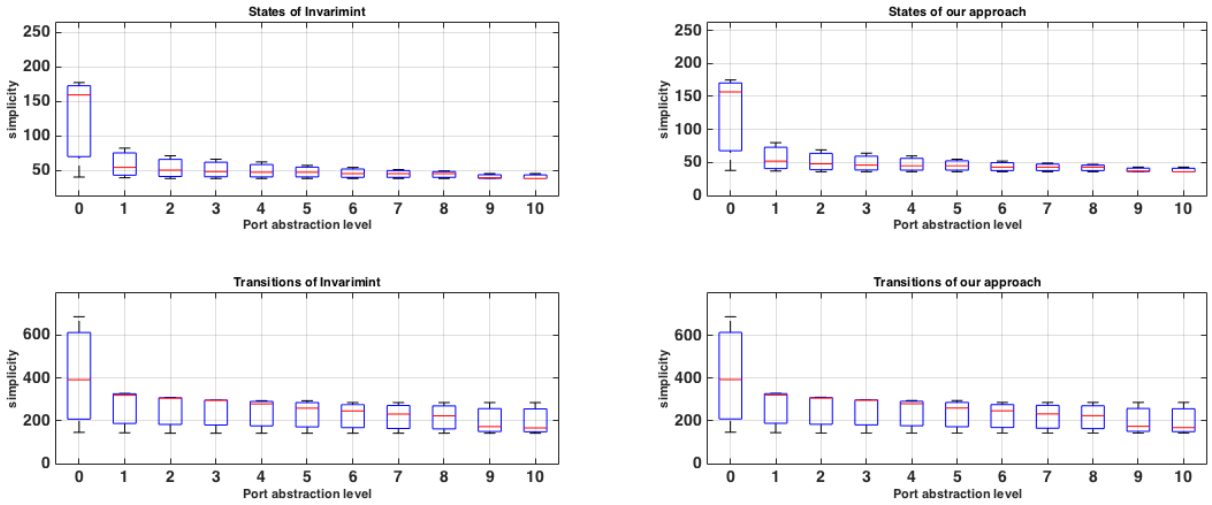

FIGURE 3.10 – Impact de l'abstraction sur les ports sur la simplicité des automates

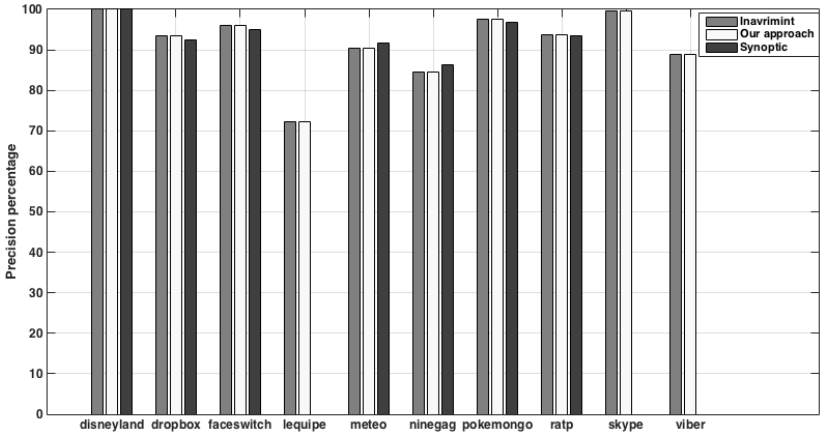

FIGURE 3.11 – Précision des automates basés sur les orgnames

3.4.3 Evaluation de la précision des automates

Les nombres élevés d'états et de transitions de ces automates sont causés par la grande hétérogénéité des fichiers de logs attachés aux applications concernées. La complexité des automates diminue plus significativement pour des valeurs seuil entre 0 et 3. Ceci peut être expliqué par le profil pair à pair des applications considérées qui conduisent beaucoup de numéros de port à apparaître seulement une ou deux fois. Les numéros de port apparaissant plus de 4 ou 5 fois peuvent être considérés comme étant représentatifs du comportement de l'application, cependant, pour des logs ou de telles valeurs pourraient constituer une part importante du traffic il pourrait être intéressant de considérer des valeurs de seuil plus importantes.

Le critère suivant que nous avons considéré est la précision des automates : étant donné une application nous définissons la précision de son automate comme le pourcentage de logs d'autres applications qu'il rejette, en d'autres termes comme le pourcentage de traffic étranger rejeté par l'automate. Avoir une haute précision est important afin de garantir que le traffic d'une application ne sera pas bloqué par les règles générées pour une autre application. D'un autre côté, si deux applications acceptent le même traffic alors leurs règles pourraient potentiellement être combinées dans un seul ensemble bénéficiant aux deux applications. Nous avons évalué la précision des automates générés par chaque méthode pour les différentes applications lorsque nous utilisions l'abstraction basée sur les orgnames : ces résultats sont présentés dans la figure 3.11.

On peut observer que la précision dépend davantage de l'application qui est considérée que de la méthode utilisée pour générer les automates. De plus pour certaines applications l'utilisation de l'abstraction basée sur les

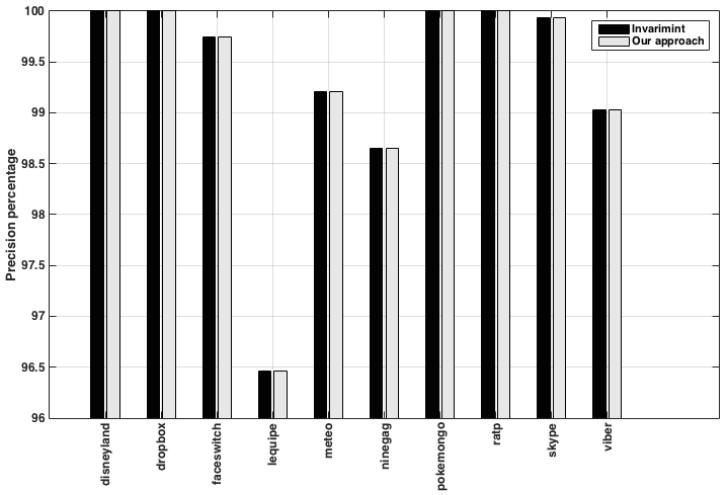

FIGURE 3.12 – Précision des automates basée sur les adresses IP

orgnames pour calculer les automates est suffisant pour avoir une précision satisfaisante. Toutefois pour d'autres telles que par exemple lequipe, la précision n'est pas satisfaisante. Cette différence est due au fait que certaines applications ne contactent que des services privés tels que par exemple disney world wide service tandis que d'autres contactent beaucoup de services publics tels que Amazon ou Google. Pour compléter ces résultats nous avons conduit une autre série d'expériences avec des automates générés à partir des adresses IP contactées par l'application : ces résultats sont présentés en figure 3.12.

A nouveau, la précision dépend davantage de l'application considérée que de la méthode utilisée pour apprendre les automates. Nous pouvons voir que la précision des automates est meilleure sans abstraire les logs [2] : Ce fait confirme notre stratégie d'utiliser les adresses IP pour reconnaître le traffic au niveau du réseau. Cependant le fort recouvrement observé avec l'utilisation de l'abstraction sur les orgnames pour générer les automates pourrait être utilisé pour repérer des sections du traffic de plusieurs applications pouvant être protégées par le même type de chaînes de fonctions de sécurité, donc l'information sur les adresses IP serait utilisée pour configurer ces chaînes.

3.5 Résumé

Dans ce chapitre nous avons présenté le résultat de nos travaux sur la conception d'un orchestrateur de chaînes de fonctions de sécurité pour des environnements intelligents. Nous avons également présenté le résultat de nos travaux liés à l'apprentissage de modèles comportementaux d'applications réseau. Nous avons commencé par discuter l'intégration d'un tel orchestrateur au sein d'un réseau SDN en coopération étroite avec le contrôleur, les switches et les équipements intelligents. Nous avons ensuite présenté l'architecture interne de notre orchestrateur et de ses différents modules. Finalement nous avons présenté le prototype implémentant cette architecture dans la pratique. Notre modèle se base sur les flux générés par une application lors de ses communications avec des services distants, nous avons eu recours à deux stratégies d'agrégation pour minimiser l'hétérogénéité de ces enregistrements. A partir de ces logs agrégés nous calculons le modèle markovien du comportement réseau d'une application. Nous avons comparé notre approche d'apprentissage avec deux méthodes utilisées dans ce domaine, Synoptic et Invarimint. Au terme de ces expériences il apparaît que notre approche permet de générer des automates plus simples que ceux fournis par Synoptic tout en étant plus expressifs que ceux fournis par Invarimint.

Ce travail constitue donc la première étape de notre processus d'orchestration de chaînes de fonctions de sécurité. Les automates que nous générons doivent ensuite être traités afin de identifier les besoins en termes de sécurité de chaque application, ces besoins seront ensuite utilisés pour construire les chaînes correspondantes. Il est en outre à noter que notre modèle pourrait encore être enrichi par l'apport de l'étude du comportement des applications à d'autres niveaux, par exemple au niveau des appels systèmes ou encore des séquences d'appels de méthodes observés au niveau fonctionnel. Quoi qu'il en soit notre modèle permet d'identifier des comportements

2. Observation sur les échelles des différents axes dans la figure 3.12.

malveillants et / ou dangereux devant être prévenus, ces points seront développés dans les prochains chapitres de ce mémoire.

4

Synthèse de chaînes de fonctions de sécurité

Nous avons retenu d'exploiter une approche de synthèse formelle de chaînes de fonctions de sécurité en vue de limiter la complexité du modèle à considérer. Dans ce contexte, nous exploitons notre modèle markovien décrit au chapitre 3 qui nous permet de construire le modèle logique d'une application puis d'inférer la chaîne correspondante. Cette approche doit être répétée pour toutes les applications à protéger, il est ensuite nécessaire de factoriser les chaînes ainsi obtenues afin de minimiser l'impact de leur déploiement sur le réseau.

Ce chapitre présente le résultat de nos travaux de synthèse de chaînes de sécurité présentés dans [89], ainsi que celui de nos travaux concernant leur factorisation présentés dans [93]. Nous commençons par décrire notre approche de construction du modèle logique d'une application, avant de détailler notre système d'inférence basé sur de la programmation logique. Nous faisons ensuite la preuve de plusieurs propriétés garanties par notre approche de synthèse. Après cela, nous motivons le problème de factorisation en l'illustrant avec deux approches naïves puis nous introduisons nos algorithmes de factorisation. Finalement, nous faisons l'évaluation des performances de notre approche de factorisation en la comparant avec les deux autres approches naïves introduites dans ce chapitre.

4.1 Construction du modèle logique d'une application

Nous commençons par décrire la construction du modèle logique d'une application à partir de flux réseau.

4.1.1 Définitions et rappels concernant les flux réseau

Rappelons que les flux réseau sont des collections de paquets partageant certains attributs. Nous résumons les flux dans des enregistrements qui contiennent les attributs clés des flux. Dans la suite, $\mathbb{N}$ et $\mathbb{R}^+$ dénotent les ensembles des entiers naturels et des réels non négatifs, $\textsc{Addr} = \{0,1\}^{32}$ et $\textsc{Port} = \{1,\ldots,65535\}$ les ensembles des adresses IP et des numéros de ports, $\textsc{Prot} = \{TCP, UDP, ARP, ICMP, \ldots\}$ l'ensemble des protocoles réseau et $\textsc{String}$ l'ensemble des chaînes de caractères (ASCII).

Definition 1. Un *flux* f est un enregistrement contenant les attributs suivants et les associant à des valeurs dans les domaines correspondants :

$$f.timestamp \in \mathbb{R}^+ \qquad f.srcaddr \in \textsc{Addr}$$
$$f.dstaddr \in \textsc{Addr} \qquad f.srcport \in \textsc{Port}$$
$$f.dstport \in \textsc{Port} \qquad f.bytes \in \mathbb{N}$$
$$f.packets \in \mathbb{N} \qquad f.protocol \in \textsc{Prot}$$
$$f.appname \in \textsc{String} \qquad f.orgname \in \textsc{String}$$

Une *trace de flux* est une suite de flux telle que les time stamps soient strictement croissants. Par abus de notation, nous écrirons $f \in t$ pour indiquer que le flux f apparaît comme un élément de la suite t. Etant données deux traces t_1 et t_2, leur *fusion* $t_1 \oplus t_2$ correspond à l'unique trace formée par les éléments de t_1 et t_2 dans l'ordre croissant des time stamps, avec la condition qu'à chaque fois que les traces t_1 et t_2 contiennent des flux f_1 et f_2

ayant même date, tel que $f_1.timestamp = f_2.timestamp$, alors f_1 apparaît dans $t_1 \oplus t_2$ tandis que f_2 est supprimé. Pour une application *app* nous noterons t_{app} une trace de flux telle que $f.appname = app$ pour tous les flux f dans la trace, et P_{app} la liste de permissions qu'elle requiert. Nous introduisons $\mathcal{D}_{danger}$ comme étant l'ensemble de permissions Android définies comme dangereuses par les spécifications du système Android. □

4.1.2 Caractérisation logique du comportement d'une application

La première étape de la construction du profil logique d'une application consiste à classifier les flux observés en accord avec les types d'attaques mentionnés au chapitre 2. Comme indiqué au chapitre 3, la trace réseau t_{app} générée par une application est traduite par une chaîne de Markov dont les états L_{app} correspondent à des sous-traces (non nécessairement contiguës) de t_{app} consistant en flux avec le même attribut *orgname*. Les transitions T_{app} de la chaîne de Markov sont des triplets (l, p, l') pour les états $l, l' \in L_{app}$ et une probabilité $p \in [0; 1]$. Observons que pour tout flux $f \in t_{app}$, il n'y a qu'un seul état $l \in L_{app}$ (correspondant à $f.orgname$) tel que $f \in l$; nous dénotons cet état comme l_f.

L'analyse des probabilités de transition apparaissant dans la chaîne de Markov, et en particulier celles associées à des boucles sur un seul état, est à la base de notre détection d'attaques telles que des dénis de service, des scans de ports ou encore du traffic de vers. En effet ces attaques sont caractérisées par une forte émission de traffic à destination d'une ou de plusieurs adresses IP appartenant à la même organisation. Il est donc possible de les découvrir dans notre automate par l'analyse des transitions bouclant sur un même état.

Nous classifions donc les flux et par extension leurs adresses de destination en nous basant sur les métriques suivantes définies pour une trace de flux t de longueur $n > 1$:

$$
\begin{aligned}
avg\_interval(t) &= \frac{\sum_{i=2}^{n} t_i.timestamp - t_{i-1}.timestamp}{n-1} \\
avg\_size(t) &= \frac{\sum_{i=1}^{n} t_i.packets}{n} \\
count(x,t) &= |\{i \in 1\,..\,n : t_i.dstaddr = x \vee t_i.dstport = x\}| \\
ports(t) &= \{p \in \textsc{Port} : \exists i \in 1\,..\,n : t_i.dstport = p\} \\
protocols(t) &= \{p \in \textsc{Prot} : \exists i \in 1\,..\,n : t_i.protocol = p\}
\end{aligned}
$$

De plus, *bgp_ranking*(*ip*) dénote une métrique correspondant à une valeur de confiance de l'adresse IP *ip*. En pratique, cette valeur est obtenue en contactant un service distant.

Nous associons les seuils suivants aux métriques définies ci-dessus ; les valeurs appropriées pour chaque seuil sont définies par les administrateurs et opérateurs réseau.

- *attack_limit* : probabilité maximale des transitions bouclant sur un même état pour considérer l'état d'un automate comme étant normal ;
- *min_interval* : minimum pour l'intervalle de temps d'arrivée entre les paquets d'un même flux ;
- *min_size* : nombre de paquets minimum dans un flux ;
- *ip_limit* : nombre d'occurrences maximal pour une adresse IP ;
- *port_limit* : nombre d'occurrences maximal pour un numéro de port ;
- *port_scan_limit* : nombre de numéros de port minimum dans une trace pour la considérer comme étant une trace de scan de ports ;
- *unsafe_threshold* : valeur maximale de *bgp_ranking* pour considérer une adresse IP comme étant sûre.

Au coeur de notre approche de détection se trouve un algorithme de classification des adresses destinations a apparaissant dans les flux de t_{app} en accord avec les prédicats suivants qui indique si a est suspectée d'être la cible d'une attaque d'un des différents types que nous considérons. Ces prédicats sont présentés à titre d'exemple d'une stratégie de détection pouvant être mise en oeuvre par l'opérateur d'un réseau, il est possible de substituer ces règles par d'autres sans modifier l'architecture globale de notre orchestrateur. Dans chacune de ces définitions nous considérons qu'il existe un flux f et une trace relative à une application t_{app} tels que :

$$
\begin{aligned}
dos(a) &\equiv f \in t_{app} \wedge a = f.dstaddr \wedge (l_f, p, l_f) \in T_{app} \wedge \\
&\quad p \geq attack\_limit \wedge count(a, l_f) \geq ip\_limit \wedge \\
&\quad avg\_interval(l_f) \leq min\_interval \wedge avg\_size(l_f) \leq min\_size \\
port\_scan(a) &\equiv f \in t_{app} \wedge a = f.dstaddr \wedge (l_f, p, l_f) \in T_{app} \wedge \\
&\quad p \geq attack\_limit \wedge count(a, l_f) \geq ip\_limit \wedge \\
&\quad avg\_interval(l_f) \leq min\_interval \wedge avg\_size(l_f) \leq min\_size \wedge \\
&\quad |\, ports(l_f) \,| \geq port\_scan\_limit \\
worm(pt) &\equiv f \in t_{app} \wedge pt = f.dstport \wedge (l_f, p, l_f) \in T_{app} \wedge \\
&\quad p \geq attack\_limit \wedge count(pt, l_f) \geq port\_limit \\
botnet(a, pt) &\equiv f \in t_{app} \wedge a = f.dstaddr \wedge (count(a, l_f) \geq ip\_limit \wedge pt = f.dstport) \vee \\
&\quad protocols(l_f) \cap \{\text{“}\mathsf{tcp}\text{”}, \text{“}\mathsf{udp}\text{”}\} \neq \emptyset \;\Rightarrow avg\_interval(l_f) \leq min\_interval \\
unsafe(a) &\equiv f \in t_{app} \wedge a = f.dstaddr \wedge bgp\_ranking(a) \geq unsafe\_threshold \\
safe(a) &\equiv \neg dos(a) \wedge \neg port\_scan(a) \wedge \neg worm(pt) \wedge \neg botnet(a, pt) \wedge \neg unsafe(a) \\
danger(pm) &\equiv pm \in P_{f.appname} \cap \mathcal{D}_{danger}
\end{aligned}
$$

En quelques mots, une adresse est considérée comme étant potentiellement la cible d'une attaque s'il existe un flux dans t_{app} pour lequel certaines valeurs seuil sont dépassées. Plus précisément $dos(a)$ exprime le fait qu'une adresse est la cible d'un déni de service si elle est la cible d'un nombre important de flux de petite taille à destination d'un numéro de port en particulier pendant un intervalle de temps restreint.

Le prédicat $port\_scan(a)$ exprime le fait que l'adresse a est la cible d'un scan de ports. Nous détectons ce type de comportement comme étant l'émission d'un grand nombre de flux de petite dimension vers plusieurs ports d'une même destination, le nombre de ports limite pour que le traffic soit considéré comme un scan étant fixé par l'opérateur du réseau.

Le prédicat $worm(pt)$ exprime le fait qu'un worm tente de se répliquer dans la trace en exploitant le port pt. Ce comportement est détecté en recherchant des scans de ports qui visent un même port sur de nombreuses adresses IP différentes au lieu de cibler balayer tous les ports d'une seule adresse IP.

Le prédicat $botnet(a, pt)$ indique quant à lui qu'une adresse a est contactée par un botnet sur le port pt. Pour assurer une telle détection nous recherchons une quantité anormale de traffic qui n'ait pas déjà été classifiée comme étant l'une des attaques précédemment caractérisées. De même si l'on observe du traffic utilisant des protocoles autres que TCP ou UDP celui-ci est automatiquement considéré comme constitutif d'un botnet, ces protocoles étant les deux principalement utilisés par les applications Android.

Enfin le prédicat $unsafe(a)$ indique qu'une adresse a sera considérée comme non sûre si jamais la valeur du ranking BGP qui lui est associée dépasse le seuil fixé par l'opérateur du réseau. Les adresses qui ne sont pas la cible d'une attaque sont considérées comme étant sûres. De plus, le prédicat *danger* enregistre des permissions déclarées par une application et considérées comme dangereuses. Par exemple, quelques propriétés dérivées pour l'application Pokemon Go à partir de son automate sont décrites dans le Listing 4.1.

```
unsafe(169.45.223.20)
unsafe(37.58.73.183)
unsafe(54.241.184.32)
unsafe(54.241.165.61)
unsafe(173.192.233.91)
```

Listing 4.1 – Exemple d'adresses contactées par l'application Pokemon Go et considérées comme étant non sûres.

4.2 Construction de la chaîne associée à une application

Nous présentons maintenant un programme basé sur des règles pour inférer la chaîne de fonctions de sécurité qui doit être déployée en réponse à une trace de flux, basé sur la classification des flux qui occurrent dans la trace. En résumé, nous commençons par associer des règles élémentaires avec les adresses qui apparaissent dans la trace. Ces règles seront ensuite composées en parallèle afin de obtenir des fonctions de sécurité telles que des pare-feux ou des systèmes de détection d'intrusion, qui seront elles-même composées en séquence afin de obtenir la chaîne

entière. Nous commençons par présenter notre approche de synthèse d'un point de vue symbolique, après quoi nous expliquerons comment cette spécification abstraite des chaînes est traduite dans le langage Pyretic.

4.2.1 Modèle logique des chaînes de fonctions de sécurité

Notre approche vise à construire une chaîne de fonctions de sécurité pour protéger une application donnée. Les fonctions de sécurité transforment le traffic réseau, i.e. des suites de paquets. Les paquets contiennent des champs d'entête similaires à ceux des flux, excepté ceux qui contiennent une information agrégée tels que par exemple *packets* et *bytes*, mais ils contiennent également une charge (le champ *payload*) qui représente l'information transmise par un paquet. Nous surchargeons l'opération de fusion $\oplus$ afin de l'appliquer à des suites de paquets de même manière qu'à des traces de flux.

Les fonctions de sécurité sont construites par composition de blocs de base, appelés *règles de sécurité*, et elles donnent lieu à des chaînes par application de compositions parallèles ou séquentielles. Les règles, fonctions et chaînes de sécurité transforment les traces de flux, en particulier en bloquant certains flux et en modifiant certaines valeurs de champs d'entête.

Definition 2. Etant donné l'ensemble $\mathcal{P}$ des paquets réseau nous définissons le traffic comme toute séquence de paquets $\mathcal{P}^*$. Une fonction de sécurité $s : \mathcal{P}^* \to \mathcal{P}^*$ apposant certaines modifications sur le trafic entrant.

Pour un entier $n \in \mathbb{N}$, la fonction $cut(t, n)$ retourne le préfixe du traffic t consistant d'au plus n paquets. Etant donné un prédicat $pred(p)$ sur les paquets, nous définissons la fonction $restrict(t, pred)$ qui retourne la sous-suite de traffic t constitué par les paquets satisfaisant $pred$.

Les fonctions de sécurité peuvent être composées en séquence ($\circ_{\gg}$) ou bien en parallèle ($\circ_{+}$) où

$$(s_1 \circ_{\gg} s_2)(t) = s_2(s_1(t))$$
$$(s_1 \circ_{+} s_2)(t) = s_1(t) \ \oplus \ s_2(t)$$

et ces opérateurs se généralisent à des compositions n-aires $\bigcirc_{\gg}$ et $\bigcirc_{+}$. □

4.2.2 Système d'inférence de la spécification logique d'une chaîne de fonctions de sécurité

Les règles de sécurité élémentaires font usage des prédicats suivants qui sont définis à part . Ceux-ci seront ensuite traduits différemment en fonction du type d'implémentation des chaînes retenu par l'opérateur du réseau. En particulier il est possible de les transcrire sous la forme de requêtes en Pyretic ou bien sous la forme de règles de VNF dans le cas où l'on ferait appel à de la virtualisation de fonctions réseau.

- $regexp(s, pm)$: vrai si la chaîne de caractères s (représentant le contenu du paquet) satisfait l'expression régulière associée avec la permission pm ;
- $tcp\_check(t)$: vrai si le traffic réseau t est une connexion TCP valide ;
- $udp\_check(t)$: vrai si le traffic réseau t est une connexion UDP valide ;
- $http\_check(s)$: vrai si la chaîne de caractères s (représentant le contenu du paquet) est une requête HTTP valide ;
- $inspect\_payload(s)$: vrai si la chaîne de caractères s (représentant le contenu du paquet) est acceptée à l'inspection profonde du paquet.

Notre système se base sur les règles de sécurité élémentaires suivantes :

$$forward(a, t) = restrict(t,\ \lambda pk : pk.dstaddr = a)$$
$$block(a, pt, t) = restrict(t,\ \lambda pk : pk.dstaddr \neq a \wedge pk.dstport \neq pt)$$
$$limit(a, n, t) = cut(forward(a, t), n)$$
$$filter(a, pm, t) = restrict(t,\ \lambda pk : pk.dstaddr = a \wedge regexp(pk.payload, pm))$$
$$inspect(a, t) = restrict(t,\ \lambda pk : pk.dstaddr = a \wedge inspect\_payload(pk.payload))$$
$$tcp(a, pt, t) = \begin{cases} restrict(t,\ \lambda pk : pk.dstaddr = a \wedge pk.dstport = pt) \\ \qquad \text{if } tcp\_check(t) \\ \langle\rangle \quad \text{otherwise} \end{cases}$$
$$udp(a, pt, t) = \begin{cases} restrict(t,\ \lambda pk : pk.dstaddr = a \wedge pk.dstport = pt) \\ \qquad \text{if } udp\_check(t) \\ \langle\rangle \quad \text{otherwise} \end{cases}$$
$$http(a, pt, t) = restrict(t,\ \lambda pk : pk.dstaddr = a \wedge pk.dstport = pt \\ \wedge\ http\_check(pk.payload))$$

Le système d'inférence présenté ci-dessous précise quelle règle associer à chaque type de prédicats logiques. Pour toutes les règles de sécurité élémentaires r présentées ci-dessus un prédicat correspondant $deploy_r$ indique si la règle doit être instanciée avec les paramètres correspondant à l'adresse IP, le numéro de port etc.

$$
\begin{aligned}
deploy_{block}(a, pt) &\leftarrow worm(pt)\\
deploy_{block}(a, pt) &\leftarrow botnet(a, pt)\\
deploy_{forward}(a) &\leftarrow \neg worm(pt) \wedge \neg botnet(a, pt)\\
deploy_{limit}(a, ip\_limit) &\leftarrow dos(a)\\
deploy_{limit}(a, ip\_limit) &\leftarrow port\_scan(a)\\
deploy_{tcp}(a, pt) &\leftarrow f \in t_{app} \wedge a = f.dstaddr \wedge pt = f.dstport \wedge f.protocol = \text{“}\mathsf{tcp}\text{”}\\
deploy_{udp}(a, pt) &\leftarrow f \in t_{app} \wedge a = f.dstaddr \wedge pt = f.dstport \wedge\\
&\quad\ pt \neq 80 \wedge pt \neq 443 \wedge f.protocol = \text{“}\mathsf{udp}\text{”}\\
deploy_{http}(a, 80) &\leftarrow f \in t_{app} \wedge a = f.dstaddr \wedge f.dstport = 80\\
deploy_{http}(a, 443) &\leftarrow f \in t_{app} \wedge a = f.dstaddr \wedge f.dstport = 443\\
deploy_{filter}(a, pm) &\leftarrow unsafe(a) \wedge\ danger(pm)\\
deploy_{inspect}(a) &\leftarrow unsafe(a)
\end{aligned}
$$

Basé sur les prédicats $deploy_r$ supposés être vrais pour une trace de traffic donnée t_{app} caractérisant le traffic réseau de l'application que nous visons à protéger, nous construisons maintenant les fonctions de sécurité par composition des règles élémentaires en parallèle.

$$
\begin{aligned}
stateless\_firewall(t) = &\quad \bigcirc_+\{\, forward(a,t) : deploy_{forward}(a),\ a \in \textsc{Addr}\,\}\\
&\circ_+ \bigcirc_+\{\, block(a,pt,t) : deploy_{block}(a,pt),\ a \in \textsc{Addr},\ pt \in \textsc{Port}\,\}\\
ids(t) = &\bigcirc_+\{\, limit(a,n,t) : deploy_{limit}(a,n),\ a \in \textsc{Addr},\ n \in \mathbb{N}\,\}\\
stateful\_firewall(t) = &\quad \bigcirc_+\{\, tcp(a,pt,t) : deploy_{tcp}(a,pt),\ a \in \textsc{Addr},\ pt \in \textsc{Port}\,\}\\
&\circ_+ \bigcirc_+\{\, udp(a,pt,t) : deploy_{udp}(a,pt),\ a \in \textsc{Addr},\ pt \in \textsc{Port}\,\}\\
&\circ_+ \bigcirc_+\{\, http(a,pt,t) : deploy_{http}(a,pt),\ a \in \textsc{Addr},\ pt \in \textsc{Port}\,\}\\
dpi(t) = &\bigcirc_+\{\, inspect(a,t) : deploy_{inspect}(a),\ a \in \textsc{Addr}\,\}\\
dlp(t) = &\bigcirc_+\{\, filter(a,pm,t) : deploy_{filter}(a,pm),\ a \in \textsc{Addr},\ pm \in \mathcal{D}\,\}
\end{aligned}
$$

4.2.3 Traduction de la chaîne en pyretic

La dernière étape de notre approche de synthèse consiste à générer le code Pyretic implantant la fonction abstraite que nous avons précédemment calculée. Ci-dessous nous fournissons les règles de réécriture pour produire l'implémentation Pyretic correspondant aux règles de sécurité élémentaires introduites ci-dessus. Dans ces règles de réécriture, l'argument des règles de sécurité correspondant au traffic t reste implicite dans la traduction Pyretic, qui est appliquée au flux de paquets courant. Les fonctions *DPIQuery*, *TCPFilter*, *UDPFilter*, and *HTTPFilter* font partie de notre valideur Synaptic (cf. chapitre 5) utilisant les politiques de requête dynamique que Pyretic fournit. La traduction de la chaîne globale est ensuite obtenue en composant le code Pyretic en séquence ou en parallèle en utilisant les combinateurs $\gg$ et $+$ de Pyretic. Les définitions suivantes indiquent l'implémentation des diverses règles de sécurité élémentaires ;

$$
\begin{aligned}
forward(a,t) &\rightsquigarrow match(dstaddr = a)\\
block(a,pt,t) &\rightsquigarrow \sim match(dstaddr = a,\ dstport = pt)\\
limit(a,n,t) &\rightsquigarrow LimitFilters(n, dstaddr = a)\\
filter(a,pm,t) &\rightsquigarrow match(dstaddr = a) \gg RegexpQuery(regexp(pm))\\
inspect(a,t) &\rightsquigarrow match(dstaddr = a) \gg DPIQuery\\
tcp(a,pt,t) &\rightsquigarrow match(dstaddr = a, dstport = pt) \gg TCPFilter\\
udp(a,pt,t) &\rightsquigarrow match(dstaddr = a, dstport = pt) \gg UDPFilter\\
http(a,pt,t) &\rightsquigarrow match(dstaddr = a, dstport = pt) \gg HTTPFilter
\end{aligned}
$$

4.2.4 Reprise de l'exemple de l'application Pokemon Go

En reprenant notre exemple avec l'application Pokemon Go, nous obtenons une chaîne contenant les fonctions de sécurité suivantes. (Nous ne présentons pas la définition générale puisque le système d'inférence génère trop de

règles de sécurité, mais nous montrons la structure générale de chaque fonction de sécurité.)

$$\begin{aligned} stateless\_firewall(t) &= forward(169.45.223.16, t) \circ_+ forward(169.45.223.20, t) \circ_+ \dots \\ stateful\_firewall(t) &= tcp(169.45.223.16, 80, t) \circ_+ tcp(169.45.223.20, 80, t) \circ_+ \dots \\ &\quad http(169.45.223.20, 80, t) \circ_+ \dots \\ dpi(t) &= inspect(169.45.223.16, t) \circ_+ inspect(169.45.223.20, t) \circ_+ \dots \end{aligned}$$

Ces fonctions de sécurité sont ensuite composées dans des chaînes pour être appliquées aux différents types de traffic réseau :

$$\begin{aligned} safe\_chain &= stateless\_firewall \circ_{\gg} stateful\_firewall \\ unsafe\_chain &= stateless\_firewall \circ_{\gg} stateful\_firewall \circ_{\gg} dpi \circ_{\gg} dlp \\ dos\_chain &= stateless\_firewall \circ_{\gg} ids \circ_{\gg} stateful\_firewall \\ port\_scan\_chain &= dos\_chain \\ worm\_chain &= stateless\_firewall \\ botnet\_chain &= stateless\_firewall \end{aligned}$$

Ces chaînes sont déployées pour filtrer le traffic générée par l'application correspondante en substituant les adresses IP dans les chaînes associées aux classes auxquelles elles appartiennent.

Pour l'application Pokemon Go, nous devons déployer la chaîne correspondant au traffic non sûr. Toutefois, étant donné qu'aucune permission dangereuse n'est déclarée dans le fichier de manifest de l'application, le composant DLP de la chaîne correspondante est trivial et peut être omis afin de minimiser la complexité algorithmique des traitements associés avec la chaîne.

Appliquons désormais les règles de traduction en Pyretic à cette spécification logique afin de illustrer la dernière étape de notre processus d'inférence. Les fonctions de sécurité introduites précédemment sont converties en la chaîne de fonctions de sécurité suivante.

$$\begin{aligned} stateless\_firewall &= match(dstaddr = 169.45.223.16) + match(dstaddr = 169.45.223.20) + \dots \\ stateful\_firewall &= match(dstaddr = 169.45.223.16, dstport = 80) \gg TCPFilter + \\ &\quad match(dstaddr = 169.45.223.20, dstport = 80) \gg TCPFilter + \dots + \\ &\quad match(dstaddr = 169.45.223.20, dstport = 80) \gg HTTPFilter + \dots \\ dpi &= match(dstaddr = 169.45.223.16) \gg DPIQuery + \\ &\quad match(dstaddr = 169.45.223.20) \gg DPIQuery + \dots \\ chain &= stateless\_firewall \gg stateful\_firewall \gg dpi \end{aligned}$$

4.3 Propriétés de correction des chaînes générées

La construction des chaînes basée sur une représentation de haut niveau garantit certaines propriétés de correction que nous discutons maintenant. Les propriétés de correction devant être assurées par l'ensemble des chaînes pourront être spécifiées par l'opérateur du réseau et être vérifiées par application de model checking ou de SMT solving, nous détaillerons ceci au chapitre 5 . Cette approche est complémentaire au travail présenté dans ce chapitre.

4.3.1 Routage des paquets

Deux propriétés désirables pour le routage des paquets dans le réseau sont l'absence de trous noirs et de boucles. Un trou noir intervient si jamais les paquets sont transmis vers un lien sur lequel aucune fonction de sécurité n'est couramment installée. Une boucle fait référence à un cycle dans les politiques de routage, de telle sorte que les paquets seront retransmis à une fonction de sécurité qui les a déjà acceptés. Nos fonctions de sécurité évitent ces problèmes par construction.

Lemme 1. *Les chaînes de fonctions de sécurité générées par l'approche présentée plus haut dans ce chapitre sont exemptes de trous noirs et de boucles.*

Démonstration. Dans notre contexte, les fonctions de sécurité sont des fonctions totales sur les suites de paquets, et elles sont construites à partir de règles de sécurité élémentaires par application de compositions séquentielles et parallèles. En particulier, chaque constituant de nos chaînes est complètement défini avant d'être utilisé, donc les trous noirs n'existent pas au niveau abstrait de la spécification des chaînes de fonctions de sécurité. De manière similaire, la définition de haut niveau des chaînes de fonctions de sécurité n'implique pas d'opérateur de point fixe ou des constructions cycliques similaires. L'algorithme de factorisation introduit dans la suite de ce chapitre n'altère pas la structure globale de nos chaînes et préserve donc ces propriétés. Finalement nous nous reposons sur la correspondance entre les chaînes abstraites et leur implémentation en Pyretic ou en Frenetic et sur la correction des traducteurs de ces langages afin de garantir que la suite de notre approche n'introduit pas de trou noir ou de boucle. □

4.3.2 Absence d'obscurcissement et cohérence

Les deux propriétés principales des chaînes de fonctions de sécurité qui intéressent notre travail sont *l'absence d'obscurcissement* et la *cohérence*. L'absence d'obscurcissement signifie qu'à chaque fois que deux règles sont composées en parallèle au sein d'une chaîne, uniquement l'une d'entre elles applique ses propriétés sur le traffic, afin de assurer qu'il n'existe pas de confusion dans le sens que deux règles pourraient être appliquées avec des résultats potentiellement conflictuels. En particulier, cette propriété implique la *cohérence* des règles, ce qui requiert qu'à chaque fois que deux règles sont appliquées, elles entraînent la même décision. La cohérence signifie que le traffic après l'application des chaînes de sécurité satisfait les contraintes de sécurité suivantes : le traffic sûr passe sans changement, tandis que le traffic potentiellement dangereux est soit bloqué soit limité dans des bornes acceptables.

Lemme 2. *Les fonctions de sécurité générées avec notre approche de synthèse garantissent l'absence d'obscurcissement.*

Démonstration. Les règles de sécurité élémentaires sont composées en parallèle dans la définition des fonctions de sécurité basiques *stateless_firewall*, *ids*, *stateful_firewall*, *dpi*, et *dlp*. La définition de *stateless_firewall* compose en parallèle les règles *forward* et *block*, qui peuvent potentiellement être en contradiction. Néanmoins ceci n'est possible qu'à condition qu'à la fois $deploy_{forward}(a)$ et $deploy_{block}(a, pt)$ soient vrais pour une adresse a et un port pt, et ceci est impossible du fait de la définition de ces prédicats.

De manière similaire, la composition parallèle des règles de sécurité *tcp*, *udp*, et *http* dans la définition de *stateful_firewall* n'est pas problématique du fait de la définition des prédicats correspondants $deploy_{tcp}$, $deploy_{udp}$, et $deploy_{http}$ qui sont mutuellement exclusifs. □

Nous montrons maintenant que nos chaînes de sécurité sont cohérentes avec les politiques de sécurité déterminées sur la base des traces t_{app} utilisées pour leur génération.

Lemme 3. *Etant donné une trace t_{app} caractérisant le traffic réseau généré par une application, la chaîne de fonctions de sécurité générée par notre approche de synthèse transmet le traffic vers les adresses IP considérées comme étant sûres sans y apporter de changement mais bloque ou limite le traffic vers les autres adresses IP.*

Démonstration. Une adresse est considérée comme étant potentiellement non sûre si t_{app} contient certains flux à destination de cette adresse classifiés comme du traffic de ver, de scan de port, de botnet ou simplement non-sûr. Le pare-feux appliqué comme première fonction de sécurité dans la chaîne bloquera directement le traffic à destination des adresses IP associées avec du traffic de vers ou de botnets.

Concernant le traffic dirigé vers des adresses associées avec des dénis de service ou des scans de ports, ce traffic passera le pare-feux sans états et sera ainsi transmis à l'IDS. Le traffic sera ainsi limité à un nombre de paquets borné par le seuil *ip_limit*, considéré comme étant acceptable par la politique de sécurité.

Pour les adresses associées avec des flux non-sûrs, i.e., du traffic réseau compromettant potentiellement la confidentialité des données sensibles d'une application, la chaîne de sécurité contient les fonctions de sécurité DPI et DLP qui vérifient le contenu des paquets. Celles-ci sont appliquées de sorte à bloquer les paquets qui satisfont les critères définis par les prédicats *regexp* (associés avec des permissions Android) et *inspect_payload*. Dans le cas où le traffic serait chiffré il devient nécessaire d'appliquer des méthodes d'inspection adaptées [96].

Le traffic à destination d'adresses considérées comme étant sûres n'est sujet qu'aux pare-feux sans état, qui le laissent passer sans y apposer de changement. □

4.4 Introduction du problème de factorisation

4.4.1 Modèle des chaînes de fonctions de sécurité dans une approche orientée objet

Avant de présenter nos algorithmes de factorisation, nous expliquons comment les chaînes sont représentées à ce niveau du processus d'orchestration et nous introduisons les notations nécessaires. Une chaîne de sécurité c correspond à un graphe dont les noeuds sont des fonctions de sécurité, par exemple des pare-feux, des systèmes de détection d'intrusion (IDS) et des systèmes de prévention de fuite de donnée (DLP) appliquées au réseau [56]. Dans la suite de ce chapitre, nous supposons par souci de simplicité qu'une chaîne donnée contient au plus une fonction de sécurité pour chacun des types que nous avons considéré jusqu'à présent, nos algorithmes préservent cette propriété. Cette hypothèse n'est cependant pas fondamentale à notre approche. Une chaîne c est caractérisée par les deux attributs $c.secFunctions$, représentant l'ensemble de fonctions de sécurité qu'elle contient, et $c.edges$ qui représente les liens entre ces fonctions de sécurité. L'opération $c.getTypes()$ retourne le type des fonctions de sécurité dans c, tandis que $c.getSecFunction(tp)$ retourne la (seule) fonction de sécurité d'une chaîne correspondant à un type donné tp.

Chaque fonction de sécurité sf est caractérisée par trois attributs : $sf.type$, $sf.rules$ et $sf.default$. L'attribut $sf.type$ indique le type d'une fonction de sécurité, tel que pare-feux sans état par exemple. L'attribut $sf.rules$ contient un dictionnaire de règles d. Chaque règle r a deux attributs $r.guard$ et $r.action$. Les actions sont indexées dans le dictionnaire sur la base des gardes correspondantes. On peut retirer l'ensemble des gardes d'un dictionnaire à partir de la méthode $d.getKeys()$. Finalement, l'attribut $sf.default$ définit une action par défaut à appliquer, lorsqu'il n'y a pas de garde qui accepte le traffic. Le comportement d'une fonction de sécurité est le suivant : le trafic entrant est analysé par les gardes des règles de la fonction de sécurité. Si une garde accepte le traffic, alors la fonction de sécurité applique l'action correspondante. Si aucune garde n'accepte le trafic entrant, alors la fonction de sécurité applique la règle par défaut. Lorsque plusieurs gardes acceptent le trafic entrant, alors l'action à appliquer est choisie sur la base d'une stratégie de priorités basée sur l'index des règles dans le dictionnaire de la fonction de sécurité. Chaque arc e est caractérisé par deux attributs correspondants à sa source $e.src$ et à sa destination $e.dst$ afin de inter-connecter les fonctions de sécurité.

Les fonctions de sécurité sont modélisées à partir du langage Pyretic avec nos travaux précédents, néanmoins elles peuvent être implantées avec des langages de programmation SDN plus modernes tel que par exemple le langage Frenetic.

4.4.2 Approches naïves pour la résolution du problème de factorisation

Dans un contexte où plusieurs applications seraient protégées en parallèle nous aurions alors plusieurs chaînes nécessitant d'être combinées comme présenté sur la figure 4.1. En effet le déploiement en parallèle de plusieurs chaînes requiert d'avoir plusieurs plans de contrôle en parallèle au niveau du contrôleur, résultant ainsi dans une grande complexité d'administration pouvant potentiellement introduire des failles dans l'architecture générale. Pour éviter cette faiblesse il est donc important de minimiser le nombre de chaînes déployées dans le réseau en les regroupant afin de obtenir une chaîne de fonctions de sécurité plus large et plus facile à gérer. Avant d'introduire notre algorithme nous présentons deux approches naïves servant de base à notre solution.

Une première approche naïve pour combiner les chaînes consiste à composer un ensemble de chaînes de fonctions de sécurité en parallèle en utilisant par exemple l'opérateur de composition + fourni par Pyretic. Nous nous référons à cette approche comme étant la combinaison parallèle de chaînes. Bien que son implémentation soit simple, cette solution génère un nombre de fonctions de sécurité qui croit linéairement en fonction du nombre d'applications à protéger. Le nombre de fonctions de sécurité aussi bien que le nombre de règles restent inchangés par rapport à l'ensemble de chaînes reçues en entrée. Le seul avantage de cette approche est qu'il conduit à obtenir une seule chaîne à gérer au niveau du réseau.

Une seconde approche, baptisée génération groupée, consiste à appliquer le système d'apprentissage comportemental ainsi que le moteur d'inférence simultanément sur toutes les applications à protéger. Etant donné un ensemble fixé de types de fonctions de sécurité, cette approche borne le nombre de fonctions de sécurité qui seront générées, du moment que l'on ne génère qu'une fonction par type. Toutefois, cette approche introduit une lourdeur supplémentaire relativement à certaines fonctions de sécurité, tel que par exemple les systèmes de prévention de fuite de données, qui combinent des informations provenant des traces réseau avec les permissions requises par les applications. Les règles correspondantes sont ainsi générées sur la base du produit cartésien des adresses IP non sûres devant être protégées ainsi que des permissions requises par les applications. Quand de multiples adresses IP ainsi que de multiples permissions demandent à être considérées, ceci conduit à un nombre de règles croissant de manière quadratique. De plus, le produit cartésien change la sémantique de la chaîne de sécurité comparée aux chaînes individuelles. Au lieu d'associer les adresses non sûres contactées par une application uniquement avec les

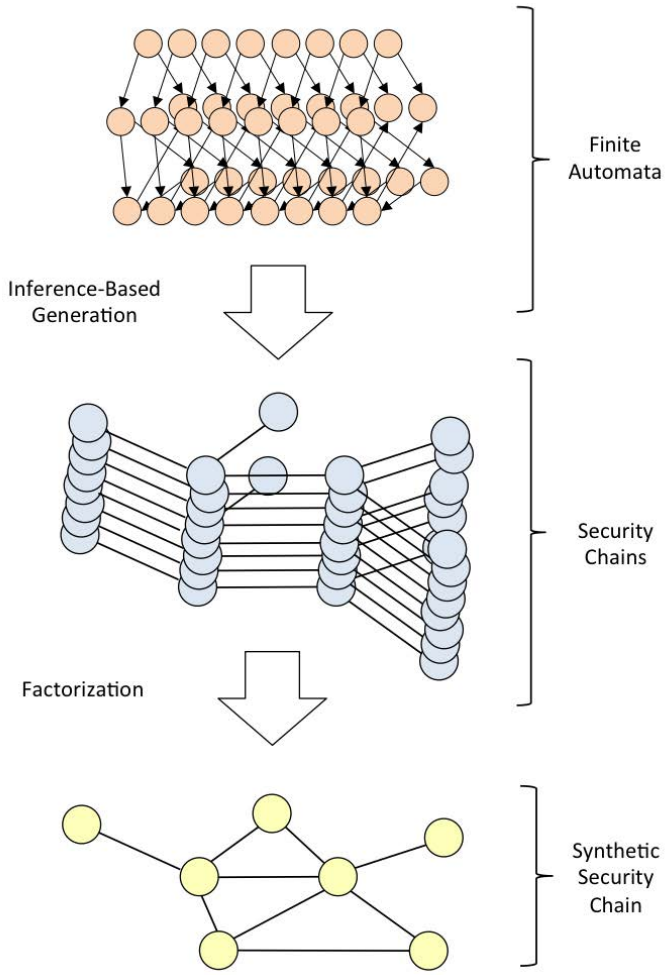

FIGURE 4.1 – Illustration du problème de factorisation

permissions qu'elle déclare, chaque adresse IP non sûre est associée avec chaque permission, ce qui résulte en une chaîne trop restrictive.

4.4.3 Exemple de factorisation de chaînes de fonctions de sécurité

Nous illustrons les chaînes obtenues par application des différentes approches précédemment introduites, en utilisant un exemple simple. Considérons trois applications Android app_1, app_2 et app_3. Le module d'apprentissage de comportement construit un modèle markovien de chaque application à partir de traces collectées par des sondes réseau et des permissions requises par chaque application. A partir de ces modèles, le générateur de chaînes infère une chaîne de sécurité, exprimée en Pyretic, pour chaque application. L'exemple montré dans la figure 4.2 correspond à la chaîne générée pour protéger la première application app_1. Des chaînes de sécurité similaires sont générées pour protéger les applications app_2 et app_3. Par souci de simplicité nous nous concentrerons sur les règles générées pour les systèmes de prévention de fuite de données (DLP) que ces chaînes de sécurité contiennent, ces règles correspondant aux éléments $R_{(4,n)}$ dont le premier indice est 4. Nous dénoterons la garde de ces fonctions de sécurité par G_i et les actions correspondantes par A_j mais ces modèles abstraits des chaînes peuvent être implémentés en Pyretic ou dans tout autre langage de programmation SDN. La composition parallèle résultera dans une unique chaîne de sécurité, dont une partie apparaît dans la figure 4.3. Cette chaîne est formée de la composition en parallèle de trois DLP notés dlp_1, dlp_2, et dlp_3. Les règles de sécurité associées avec chaque fonction de sécurité restent inchangées par rapport à l'ensemble de chaînes de fonctions de sécurité initiales. Dans

$$
\begin{aligned}
stl\_fw &= R_{(1,1)} + R_{(1,2)} + R_{(1,3)} + \ldots + R_{(1,n_1)} \\
ids &= R_{2,1)} + R_{(2,2)} + R_{(2,3)} + \ldots + R_{(2,n_2)} \\
stf\_fw &= R_{(3,1)} + R_{(3,2)} + R_{(3,3)} + \ldots + R_{(3,n_3)} \\
&\quad + match(dstip = 45.67.89.123) \gg \\
&\quad (TCPFilter + HTTPFilter) \\
dlp &= R_{(4,1)} + R_{(4,2)} + R_{(4,3)} + \ldots + R_{(4,n_4)} \\
dpi &= R_{(5,1)} + R_{(5,2)} + R_{(5,3)} + \ldots + R_{(5,n_5)} \\
chain &= stl\_fw \gg ids \gg stf\_fw \gg dlp \gg dpi
\end{aligned}
$$

FIGURE 4.2 – Exemple d'une chaîne de sécurité générée pour protéger une application Android donnée

cet exemple on obtient une chaîne comprenant 3 fonctions de sécurité et 10 règles de sécurité. [3] Comme nous l'expliquions précédemment, la composition groupée implique de construire le produit cartésien des adresses IP et des permissions pour construire la chaîne combinée. Dans notre exemple, en supposant que les adresses IP et les permissions des différentes applications sont distinctes, on obtient alors une unique fonction DLP contenant 18 règles de sécurité.

4.5 Algorithme de factorisation

L'objectif de la factorisation de chaînes de sécurité est de transformer un ensemble de telles chaînes en une seule. Cette opération vise à minimiser le nombre total de fonctions de sécurité devant être déployées dans le réseau pour protéger un équipement Android ainsi que ses applications, mais également à minimiser le nombre de règles utilisées pour implémenter ces fonctions de sécurité. En contraste avec les deux approches de base présentées dans la section précédente, notre approche de composition consiste à factoriser les chaînes de fonctions de sécurité après les avoir générées. Elle est basée sur deux algorithmes : *merge_functions* (Algorithme 2) produit une unique fonction de sécurité à partir de deux fonctions données en entrée, tandis que *merge_chains* (Algorithme 3) factorise deux chaînes de fonctions de sécurité entières. Afin de gérer de potentiels conflits entre les règles de deux fonctions de sécurité appliquant des actions contradictoires au même trafic réseau, nous supposons un système de priorités entre les règles, abstraitement représenté par l'opérateur $\leq_r$ où les règles considérées comme étant les plus petites ont la plus haute priorité. Différentes implémentations de $\leq_r$ peuvent être implantées par les opérateurs réseau, tel que par exemple en prenant en compte le niveau de sûreté des adresses IP contactées par une application.

3. on observe que la composition parallèle conduit à des chaînes contenant plusieurs occurrences de la même fonction de sécurité.

$$
\begin{aligned}
dlp_1 &= G_1 \gg A_1 + G_2 \gg A_1 \\
dlp_2 &= G_1 \gg A_1 + G_1 \gg A_2 + \\
&\quad G_3 \gg A_1 + G_3 \gg A_2 + \\
&\quad G_4 \gg A_1 + G_4 \gg A_2 \\
dlp_3 &= G_5 \gg A_3 + G_6 \gg A_3
\end{aligned}
$$

FIGURE 4.3 – Extrait d'une chaîne de sécurité résultant d'une composition parallèle

Algorithme 2 Factorisation des fonctions de sécurité

```
function MERGE_FUNCTIONS(sf1, sf2 : SecFunction)
                    ▷ Construire l'ensemble des règles n'apparaissant que dans une seule fonction
    guards1 := sf1.getKeys()
    guards2 := sf2.getKeys()
    rules := new Dictionary()
    for g ∈ guards1 \ guards2 do
        rules.put(g, sf1.rules.get(g))
    end for
    for g ∈ guards2 \ guards1 do
        rules.put(g, sf2.rules.get(g))
    end for
                                          ▷ Ajouter les règles traitant le même trafic réseau
    for g ∈ guards1 ∩ guards2 do
        r1 := sf1.rules.get(g)
        r2 := sf2.rules.get(g)
        if r1 ≤r r2 then
            rules.put(g, r1)
        else
            rules.put(g, r2)
        end if
    end for
     return new SecFunction(sf1.type, rules)
end function
```

4.5.1 Factorisation des fonctions de sécurité

L'algorithme 2 reçoit deux fonctions de sécurité sf_1 et sf_2 (supposées être du même type) comme entrée. Il récupère tout d'abord les ensembles de règles de sf_1 et sf_2 et construit leur intersection ainsi que leur différence symétrique. Les gardes n'ayant pas d'équivalent dans l'autre fonction ne peuvent pas être en conflit. Elles sont donc simplement ajoutées à la fonction résultante. Pour les gardes apparaissant dans les deux fonctions de sécurité, l'opérateur de priorité $\leq_r$ est utilisé pour déterminer quelle action doit être associée avec la garde dans la fonction résultante.

4.5.2 Factorisation des chaînes

L'algorithme 3 compose deux chaînes de sécurité c_1 et c_2 en se reposant sur l'algorithme 2. Il commence par identifier les fonctions de sécurité ayant le même type dans les deux chaînes. Il applique ensuite l'algorithme 2 afin de composer ces fonctions. Les fonctions de sécurité d'un certain type n'ayant pas d'équivalent dans l'autre chaîne sont simplement ajoutées à la chaîne résultante. Finalement, les arcs de la chaîne résultante sont ajoutés à partir des liens unissant les différentes fonctions de sécurité des deux chaînes d'entrée. En appliquant successivement l'algorithme 3 aux chaînes générées pour un ensemble de fonctions de sécurité, on obtient une seule chaîne permettant de protéger toutes les applications.

4.5.3 Reprise de l'exemple de factorisation

En reprenant l'exemple de composition développé à la section précédente, on observe que composer les chaînes en utilisant l'algorithme 3 factorise à la fois les fonctions de sécurité et leurs règles constituantes. La fonction DLP de la chaîne résultante est montrée dans la figure 4.4 : On obtient une seule fonction contenant 9 règles, ce qui est plus compact que le résultat des deux autres approches. En outre il est à signaler que notre approche préserve l'intégrité de chaque DLP, contrairement à l'approche de génération groupée qui modifiait la sémantique des fonctions de sécurité en calculant le produit cartésien des adresses et des permissions. Pour ce faire notre

$$
\begin{aligned}
dlp \quad = \quad & G_1 \gg A_1 + G_2 \gg A_1 + \\
& G_1 \gg A_2 + G_3 \gg A_1 + \\
& G_3 \gg A_2 + G_4 \gg A_1 + \\
& G_4 \gg A_2 + G_5 \gg A_3 + \\
& G_6 \gg A_3
\end{aligned}
$$

FIGURE 4.4 – Extrait d'une chaîne de sécurité obtenue par application de la composition par factorisation

applications	# sf	# rules
disneyland	4	44
dropbox	5	311
faceswitch	5	425
lequipe	4	1640
meteo	4	716
ninegag	4	930
pokemongo	5	485
ratp	4	28
skype	5	6529
viber	5	4163

FIGURE 4.5 – Caractéristiques des chaînes de fonctions de sécurité générées pour chaque application.

approche compose les DLP d'une manière analogue à la composition parallèle tout en réduisant le nombre total de fonctions de sécurité.

4.6 Prototype et résultats expérimentaux

Nous avons évalué les performances de notre approche au travers d'une série d'expérimentations. Le contexte expérimental était basé sur un ordinateur portable MacBookPro avec un processeur Intel Core i7 (2.5 GHz) et 16 GB de RAM. Pendant ces expérimentations nous avons considéré l'ensemble de fichiers de logs présentés au chapitre 3. Les caractéristiques des chaînes associées avec ces applications sont présentées en fig. 4.5. Ces résultats illustrent clairement la très haute disparité du nombre de règles de sécurité générées pour chaque application. Le nombre de fonctions de sécurité varie entre 4 et 5, ceci est respectivement dû à l'absence et à la présence du fichier de manifest qui permet de calculer les règles pour le DLP à partir des permissions qui y sont déclarées. Afin de comparer les performances des approches de composition, nous avons considéré les critères d'évaluation suivants :

- la complexité des chaînes de sécurité, mesurée en terme de nombre de fonctions et de règles ;
- le surplus introduit par le système de factorisation mesuré en terme de temps de réponse ;
- la précision de détection des chaînes de sécurité.

4.6.1 Prototypage

Les classes implantant les travaux présentés dans ce chapitre correspondent aux modules PropertyLearner, et ChainGenerator présentés dans le diagramme 3.8. Ces modules représentent respectivement un total de 333 et 135 lignes de code Python, notre module d'inférence de chaînes a été implanté en SWI-Prolog (version 7.6.4) pour un total de 106 lignes de code. La spécification logique d'une application est obtenue par dérivation de l'automate markovien d'une application, les classes de prédicats correspondantes sont décrites dans le graphique 4.6.

La phase de génération de chaînes de fonctions de sécurité s'appuie sur les classes décrites dans le diagramme 4.7 pour un total de 268 lignes de code. Les classes héritant de ChainFactory implantent le design

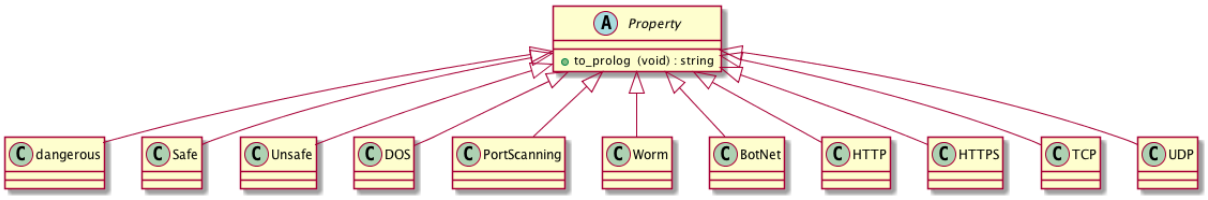

FIGURE 4.6 – Diagramme des classes du langage de propriétés utilisé par notre orchestrateur

pattern factory afin de permettre une génération des chaînes qui ne soit pas dépendante du langage retenu pour les implémenter. En effet pendant la majeure partie de la thèse nous nous sommes concentrés sur le langage Pyretic qui s'est révélé être obsolète vers la fin de nos travaux. Pour permettre l'évaluation pratique de notre approche, nous avons donc eu recours au langage de spécification netcat qui peut être utilisé en lien avec le langage Frenetic. Les langages netcat et Frenetic appartiennent à la même famille de langages que Pyretic, toutefois ils sont toujours soutenus aujourd'hui et permettent donc d'assurer le déploiement concret de nos chaînes dans le réseau.

Les classes décrivant l'implantation des chaînes et des fonctions de sécurité sont décrites dans le graphique 4.8 et représentent un total de 1577 lignes de code. Chacune des classes concrètes décrites dans ce diagramme est mère de deux sous-classes pour les implantations en Pyretic et netcat. Pour des raisons de lisibilité nous n'avons pas décrit les classes correspondantes. Cette architecture de classes rend notre approche d'orchestration agnostique quant au langage utilisé pour implémenter les chaînes, il suffit d'étendre les classes correspondantes pour intégrer une nouvelle implémentation. Le total de code Python2, pour les travaux de ce chapitre, génération et factorisation, revient à 2162 lignes de code, incluant la définition des trois approches de composition.

4.6.2 Complexité des chaînes de sécurité

Dans une première série d'expérimentations nous nous sommes intéressés à évaluer la complexité des chaînes de sécurité, en termes de nombre de fonctions et de règles de sécurité. Nous avons comparé les trois approches : la composition parallèle qui compose simplement les chaînes après leur génération, la composition groupée qui génère directement une chaîne de sécurité pour un ensemble d'applications devant être protégées, et finalement la composition par factorisation dans laquelle les chaînes sont générées individuellement puis composées par application de l'algorithme présenté dans la section précédente. Les résultats obtenus quant au nombre de fonctions de sécurité sont présentés dans la figure 4.6.3, affichant le nombre total de règles de sécurité pour chacune des trois approches de composition. Les courbes correspondant aux composition en parallèle et par factorisation se sont avérées être confondues pendant ces expérimentations, elles devraient différer dans le cas général, en fonction du niveau de redondance entre les différentes règles de sécurité. Comme espéré, on peut constater que le nombre de règles de sécurité pour la première et la troisième approche (les approches de composition parallèle et par factorisation) croissent linéairement en fonction du nombre d'applications à protéger, tandis que ce nombre croit de manière quadratique avec la seconde approche (l'approche de composition groupée). Comme mentionné plus haut, ce phénomène peut être expliqué par le fait que certaines règles, par exemples celles qui sont relatives au système de prévention de fuite de données, reposent sur le produit cartésien des adresses IP ainsi que des applications relatives aux différentes applications à protéger. Le nombre de fonctions de sécurité (non présenté dans la figure) croit linéairement avec la composition parallèle mais reste constant avec les approches de composition groupée ou bien par factorisation. Ces expérimentations illustrent clairement le bénéfice introduit par l'utilisation de l'approche de composition par factorisation pour réduire le nombre de fonctions et de règles de sécurité dans la chaîne résultante.

4.6.3 Surcoût de la composition de chaînes

Dans une seconde série d'expérimentations nous quantifions le surcoût du système proposé en termes de temps de réponse. L'objectif est de mesurer la faisabilité de l'approche en pratique.

Ces résultats ne comprennent pas le temps nécessaire à l'apprentissage du modèle markovien des applications à protéger mais ils comprennent le temps pris par le moteur d'inférence. Les temps de réponse des trois approches de composition ne sont pas présentés ici, néanmoins on peut constater que le surcoût introduit par l'approche de factorisation comparée aux approches naïves est négligeable. La composition groupée apporte des performances légèrement meilleures que les deux autres approches, ceci est dû au fait qu'il n'y a qu'une seule chaîne générée pour l'ensemble d'applications à protéger. Le point négatif de l'approche groupée est qu'elle requiert une phase

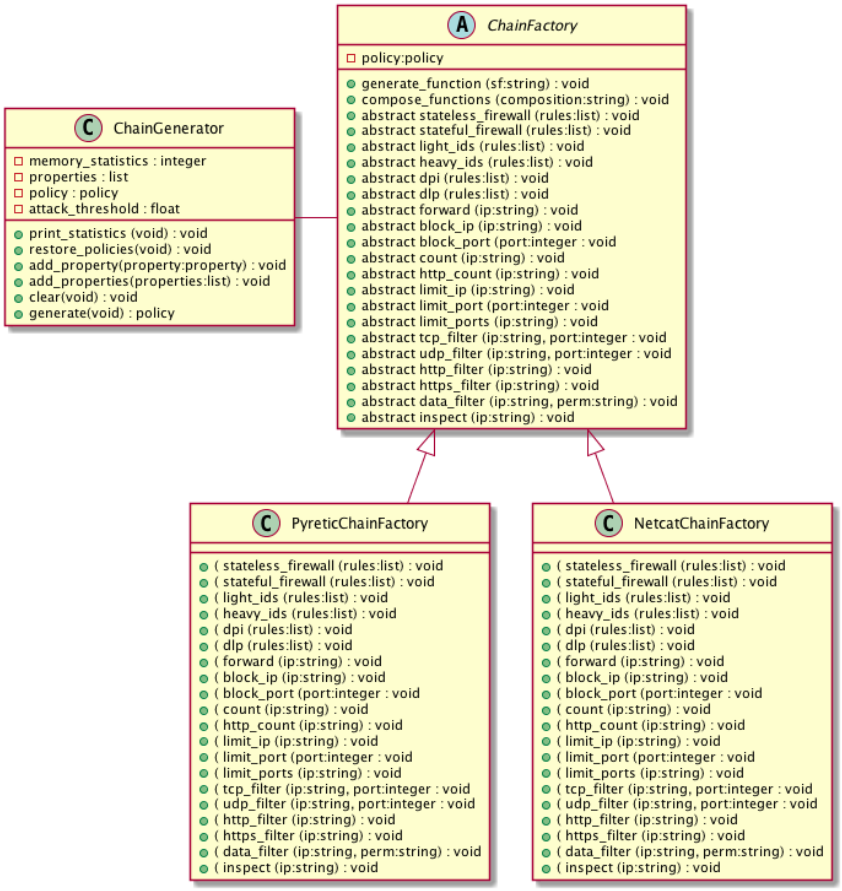

FIGURE 4.7 – Diagramme des classes du module de génération de chaînes de fonctions de sécurité

d'apprentissage juste avant la phase de génération. En supposant que les chaînes générées pour chaque application puissent être retirées d'une base de données, une comparaison plus réaliste des temps de réponse apportés par les différentes approches apparaît dans la figure 4.10 : Le temps total demandé par la phase d'apprentissage précédant l'approche de composition groupée domine le temps de réponse des deux approches par plus de deux ordres de magnitude.

Ces résultats illustrent clairement le fait qu'apprendre les modèles markoviens en temps réel n'est pas faisable. Dans notre architecture globale, nous proposons que les automates markoviens et les chaînes de sécurité soient calculés pro-activement et enregistrés dans une base de données. En fonction des applications actives sur l'équipement, les chaînes de sécurité correspondantes sont chargées dynamiquement et factorisées ensemble avant d'être déployées par le contrôleur du réseau SDN. En supposant que les applications sont relativement stables pendant une longue période de temps, le coût de l'apprentissage peut ainsi être amorti par plusieurs phases de déploiement.

4.6.4 Précision des chaînes de sécurité

En complément à ces expérimentations, nous avons évalué la précision des différentes chaînes de fonctions de sécurité. Nous avons évalué cette métrique en injectant un simple scan de ports de 50 flux dans le fichier de log de chaque application, et nous avons quantifié la précision comme le ratio de la somme des vrais positifs et vrais négatifs par le nombre total de flux. Concrètement, nous avons utilisé 70% des logs pour générer les chaînes et 30% pour l'évaluation. Nous avons également fixé un ratio de détection correspondant au nombre de flux d'attaque qui doivent être détectés avant de bloquer le trafic et nous avons fait varier ce ratio de 0 à 10 ; les résultats correspondants apparaissent dans la figure 4.11 où nous présentons la précision minimale, maximale et moyenne mesurée pour chaque chaîne de fonctions de sécurité.

Nous souhaitions également évaluer l'impact qu'avait notre approche de factorisation sur les performances

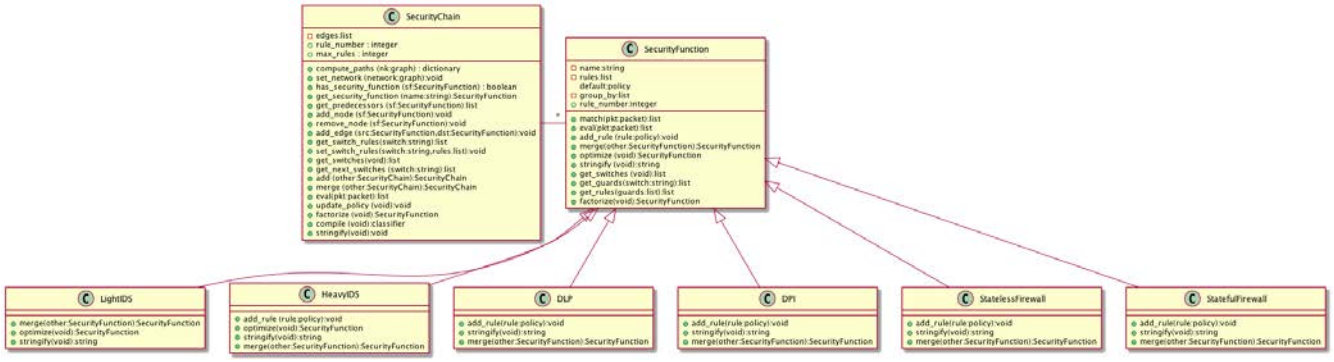

FIGURE 4.8 – Diagramme des classes implantant les chaînes et les fonctions de sécurité

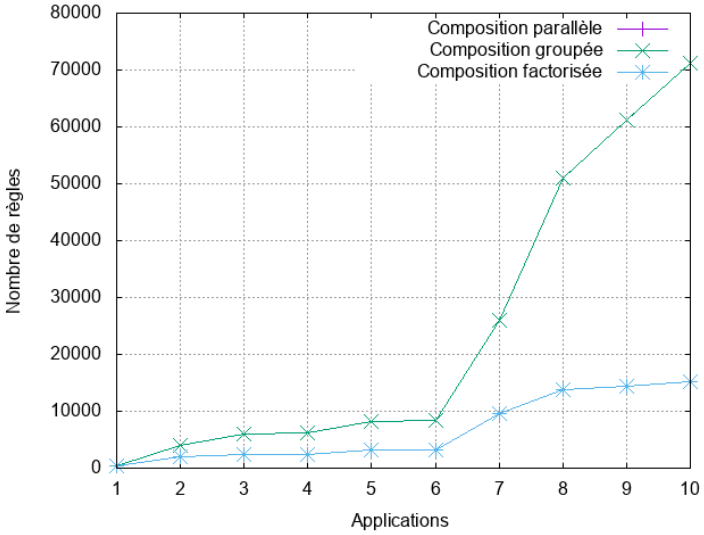

FIGURE 4.9 – Nombre de règles des différentes approches de composition

de détection de chaque chaîne, en particulier nous souhaitions mesurer si notre approche entraînait une perte de précision des chaînes individuelles. Nous avons finalement constaté que ce n'était pas le cas, en particulier nous pensions que les performances de détection seraient moins bonnes avec l'approche de composition groupée mais nous avons constaté que tel n'était pas le cas. Ceci est vraisemblablement dû au fait que dans le cas de nos expérimentations basées sur un modèle synthétique du trafic, la modélisation des données transmises n'intervient pas, d'où la permissivité des règles générés pour les DLP. Finalement nous présentons les résultats obtenus pour chaque application considérée individuellement, ces résultats sont maintenus dans le cas où ces chaînes sont composées de quelque manière que ce soit.

Nous observons de nouveau une grande disparité dans les résultats obtenus pour chaque application : pour certaines applications le taux de détection est très proche de 100% tandis que d'autres ont une précision minimale plus basse que 50%. Ceci est de nouveau causé par la nature des différents fichiers de logs : pour certaines applications les 30% de logs utilisés pour l'évaluation ne contiennent que des adresses IP qui étaient déjà connues pendant la phase d'apprentissage et sont donc de fait acceptées par la chaîne tandis que d'autres applications ont une plus grande disparité dans leur fichier de logs. L'amélioration de ces travaux demandera des collaborations avec des chercheurs travaillant dans le domaine de la détection d'attaques pour concevoir des méthodes de détection plus élaborées. Notre approche de synthèse découple cette phase de l'inférence des chaînes au travers de prédicats de la logique du premier ordre, et ceci rend facile le changement de méthode de détection d'attaques sans modifier l'approche générale.

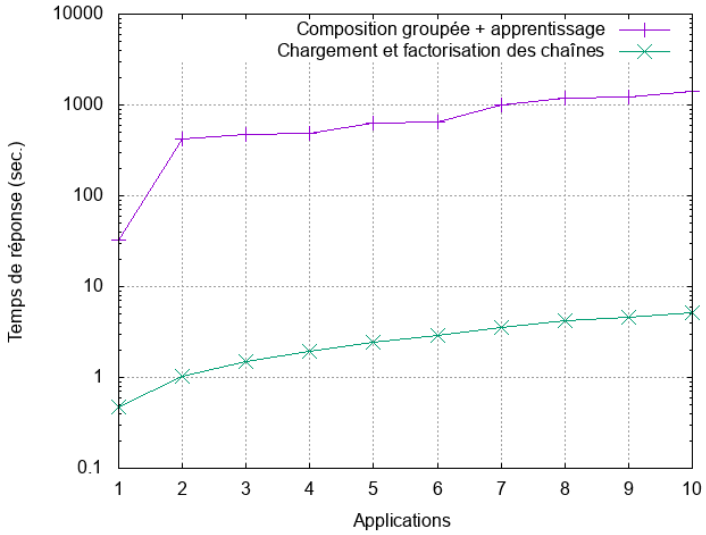

FIGURE 4.10 – Temps total pour les approches de composition groupée et par factorisation

Application	Prec. Moy.	Prec. Min.	Prec. Max.
viber	0.683	0.502	0.997
faceswitch	0.812	0.518	0.990
dropbox	0.997	0.993	1.000
ninegag	0.509	0.498	0.526
disneyland	0.992	0.986	1.000
pokemongo	0.743	0.512	0.994
skype	0.998	0.998	0.998
lequipe	0.518	0.496	0.537
meteo	0.837	0.510	0.998
ratp	0.940	0.692	0.999

FIGURE 4.11 – Précision de la chaîne générée pour chaque application

4.7 Résumé

Dans ce chapitre nous avons présenté nos travaux relatifs à la synthèse des chaînes de fonctions de sécurité et à leur factorisation. Nous avons commencé par présenter notre approche pour la construction du modèle logique d'une application, puis notre système d'inférence permettant de construire la chaîne correspondante par application de programmation logique. Après cela nous avons motivé le problème de factorisation de chaînes en l'illustrant avec deux approches naïves, puis nous avons introduit nos algorithmes pour sa résolution. Finalement, nous avons fait la preuve de plusieurs propriétés garanties par notre système d'inférence et nous avons évalué les performances de notre approche de synthèse. Au terme de ces expérimentations, il ressort que notre approche permet de construire des chaînes ayant aussi peu de règles que la composition parallèle et aussi peu de fonctions que l'approche groupée.

Les performances de détection de notre système pourraient encore être améliorées. En particulier, la prise en considération de logiques plus expressives telles que par exemple les logiques temporelles pourraient permettre une meilleure caractérisation logique du comportement des applications, et donc des chaînes associées. Concernant notre approche de factorisation, celle-ci résout le problème de la multiplicité des chaînes à administrer, en revanche elle ne résout pas le problème de la vérification de leur compatibilité, ce point sera développé dans le chapitre suivant.

Algorithme 3 Factorisation de chaînes de sécurité

```
function MERGE_CHAINS(c1, c2 : SecChain)
    res := new SecChain()
                                  ▷ Factoriser les fonctions de sécurité ayant le même type
    for sf1 ∈ c1.secFunctions, sf2 ∈ c2.secFunctions do
        if sf1.type = sf2.type then
            sf := MERGE_FUNCTIONS(sf1, sf2)
            res.secFunctions.add(sf)
        end if
    end for
                                  ▷ Ajouter les fonctions restantes à la chaîne résultante
    for sf ∈ c1.secFunctions \ res.secFunctions do
        if sf.type ∉ c2.getTypes() then
            res.secFunctions.add(sf)
        end if
    end for
    for sf ∈ c2.secFunctions \ res.secFunctions do
        if sf.type ∉ c1.getTypes() then
            res.secFunctions.add(sf)
        end if
    end for
                  ▷ Ajouter les arcs de connexion correspondant à la chaîne résultante
    for cn ∈ c1.edges do
        src := res.getSecFunction(cn.src.type)
        dst := res.getSecFunction(cn.dst.type)
        cn := new Edge(src, dst)
        res.edges.add(cn)
    end for
    for cn ∈ c2.edges do
        src := res.getSecFunction(cn.src.type)
        dst := res.getSecFunction(cn.dst.type)
        cn := new Edge(src, dst)
        res.edges.add(cn)
    end for
     return res
end function
```

5

Vérification automatique de chaînes de fonctions de sécurité

Dans ce chapitre sont présentés les résultats que nous avons obtenus en terme de vérification automatique de chaînes de fonctions de sécurité avec Synaptic. Ce framework a été développé comme une extension de Pyretic et de Kinetic avec une procédure de vérification pour le plan de données. Nous commençons par présenter l'architecture soutenant la vérification formelle des chaînes sur la base de méthodes de traduction en modèles formels. Nous présentons ensuite les algorithmes de réécriture de chaînes spécifiées en Pyretic en SMTlib ou dans le langage d'entrée de nuXmv. Nous concluons enfin avec les résultats expérimentaux que nous avons obtenus dans le cadre de ce travail.

5.1 Approche pour la vérification formelle de chaînes de fonctions de sécurité

Nous commençons par détailler la stratégie, ainsi que l'architecture considérée pour notre approche.

5.1.1 Stratégie de vérification

Nous proposons une approche de vérification des chaînes de fonctions de sécurité couvrant tout à la fois les plans de contrôle et de données. Comme nous en avons déjà longuement discuté dans le chapitre 2, la plupart des approches de vérification formelle de politiques de sécurité SDN se concentrent sur la vérification du plan de données et délaissent les aspects dynamiques liés au plan de contrôle [11, 61, 3, 46]. Ces approches laissent donc passer d'éventuelles attaques visant le déploiement de règles incorrectes au travers de l'exploitation de failles potentielles des politiques de contrôle mises en place dans les réseaux. C'est pour traiter cette limite que nous avons choisi de nous intéresser à une approche validant l'intégrité des deux plans du paradigme SDN.

Notre solution étend le framework fourni par le langage Pyretic [44]. Plus précisément le langage Pyretic est utilisé pour décrire le plan de données des chaînes de fonctions de sécurité. Une fois validées ces spécifications peuvent être converties en politiques OpenFlow avant d'être déployées dans le réseau, ceci en vue de permettre un déploiement dynamique des chaînes. De plus, l'extension Kinetic fait exception parmi les approches de vérification en couvrant le plan de contrôle et non le plan de données. En particulier, Kinetic permet la spécification du plan de données sous la forme d'un automate à états finis exprimant la dynamique du plan de contrôle en réponse à différents évènements affectant l'exécution du réseau. Plus précisément cette extension fait usage du model checker NuSMV pour vérifier si l'automate correspondant garantit des propriétés exprimées dans la logique temporelle CTL.

Nous avons donc retenu d'étendre l'approche de vérification du plan de contrôle proposée par Kinetic avec une procédure de vérification pour le plan de données. Pour ce faire nous avons choisi d'avoir recours à des méthodes de traduction automatique pour produire un modèle formel des chaînes spécifiées en Pyretic. Nous appliquons ensuite des outils de vérification formelle sur le modèle ainsi obtenu pour en assurer la cohérence. Plus précisément nous appliquons toujours la procédure de vérification fournie par Kinetic pour vérifier la validité du plan de contrôle et une fois que cette première étape est passée nous appliquons notre méthode de vérification sur chacune des politiques réseau constituant les états de cet automate.

En particulier, nous avons développé un langage complet permettant la spécification de propriétés devant être garanties par les chaînes de fonctions de sécurité. Nous nous sommes proposés de couvrir la spécification tant du plan de contrôle que du plan de données dans la conception de ce langage. Pour ce qui est du plan de contrôle nous avons étendu la spécification de propriétés en logique CTL supportée par Kinetic. Pour ce qui est du plan de données, notre langage se base sur les champs d'entête des flux analysés par une chaîne de fonctions de sécurité. Pour ce qui est des actions à effectuer en réponse à ces champs d'entête nous nous sommes restreints à considérer que les paquets sont supprimés ou transmis, le déploiement d'actions plus élaborées ayant été traitée au chapitre 4. Nous nous sommes également restreints à l'expression de propriétés sur le comportement réseau des usagers, l'investigation du contenu des données transmises dans les paquets reste à développer dans le cadre de travaux ultérieurs.

Pour donner quelques exemples des propriétés que nous considérons, celles-ci peuvent prendre la forme de conjonctions ou de disjonctions de conditions sur les champs d'entête tel que par exemple $dstport = 4000 \wedge dstip = 132.34.0.0/16$. La prise en compte de propriétés relatives au plan de contrôle se fait au travers de la logique CTL supportée par Kinetic. Par exemple, nous pouvons spécifier des propriétés telles que $AG\ infected \rightarrow AG\ policy = drop$ qui met en lien l'évènement *infected* caractérisant la détection d'une infection avec le déploiement de la politique Pyretic *drop* qui assurera que tous les paquets sont désormais supprimés pour cet hôte. Le lien entre ces deux niveaux de spécification reste à charge de l'utilisateur de notre framework qui doit veiller à ce que la spécification de ses propriétés sur le plan de données restent cohérentes avec la logique de sa politique du plan de contrôle. Pour ce faire, nous avons introduit la possibilité d'associer une politique aux propriétés de notre langage de spécification de propriétés.

5.1.2 Architecture supportant la vérification

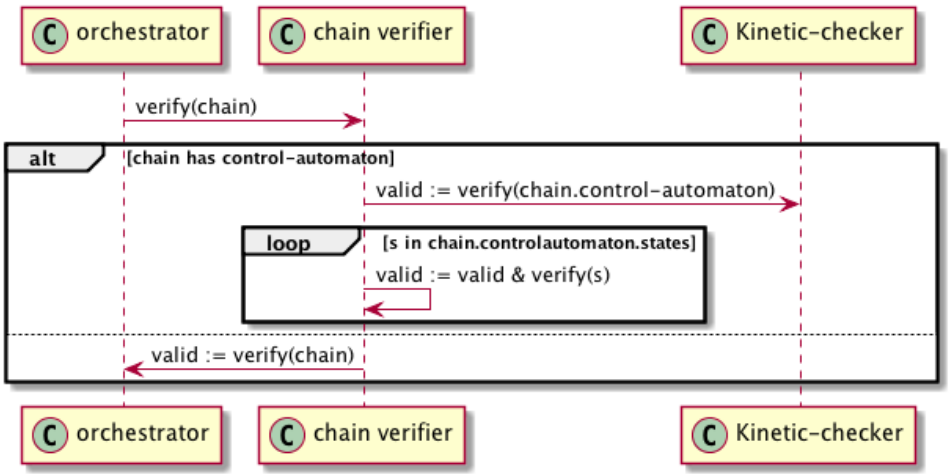

FIGURE 5.1 – Diagramme d'interactions entre les différents composants de notre module de vérification

Notre module de vérification, nommé Synaptic, implémente notre stratégie de traduction des chaînes Pyretic vers des spécifications abstraites. Nous avons considéré deux catégories de vérification formelle pour assurer la validation du plan de données d'une chaîne de fonctions de sécurité. La première catégorie correspond aux outils de SMT solving. Pour rappel, ces méthodes reposent sur une représentation logique des systèmes à vérifier ainsi que de leurs contraintes. Cette représentation contient des variables dont la valeur reste à définir pendant la procédure de vérification. Ce modèle est transmis à un SMT solver qui cherche alors un arrangement des variables de la spécification rendant toutes les contraintes vraies. La seconde catégorie de méthodes que nous avons considérées sont les méthodes de model checking. Dans ce cas, les règles sont représentées comme un automate à états finis, et les propriétés sont spécifiées avec une logique, possiblement temporelle, similaire à celle utilisée pour les propriétés du plan de contrôle. A partir de cette représentation, un model checker construit l'ensemble d'états atteignables de l'automate et vérifie la validité de ses contraintes logiques, construisant un contrexemple lorsque la propriété n'est pas respectée.

La figure 5.1 illustre l'interaction des composants dans notre architecture. Les chaînes de fonctions de sécurité spécifiées en Pyretic sont transmises par l'orchestrateur au module de vérification. Elles sont d'abord vérifiées

par application de l'extension Kinetic pour identifier des incohérences dans le plan de contrôle. Le module de vérification traduit ensuite la spécification des chaînes en un modèle formel, qui est ensuite interprété par un SMT solver ou un model checker. Les résultats de la validation sont ensuite retransmis à l'orchestrateur, afin de déterminer si les chaînes peuvent être déployées ou non dans le réseau.

5.2 Traduction des chaînes en modèles formels

Nous présentons ici les algorithmes de réécriture des chaînes de fonctions de sécurité en spécifications formelles. Nous avons retenu deux approches de traduction, l'une vers SMTlib et l'autre vers le langage d'entrée de nuXmv.

5.2.1 Algorithme de traduction vers un modèle vérifiable par SMT solving

En premier lieu, nous expliquons l'algorithme de traduction vers SMTlib, le langage d'entrée des SMT solvers. Soit x un entête de paquet, x_after le même champ après modification, y une valeur et p_1 et p_2 deux politiques données. La fonction de traduction de Pyretic vers SMTlib, dénotée par f, est définie de manière très intuitive, basée sur la grammaire de Pyretic, en trois étapes principales :

1. Traduction des règles élémentaires de Pyretic introduites au chapitre 2 en propositions atomiques, comme suit :
   — $f(identity) \rightarrow true$ and $f(drop) \rightarrow false$,
   — $f(match(x = y)) \rightarrow (= \ x \ y)$,
   — $f(modify(x = y)) \rightarrow (= \ x\_after \ y)$.
2. Traduction des opérateurs de composition (séquence, parallèle et négation) en opérateurs booléens comme suit :
   — $f(p_1 \gg p_2) \rightarrow f(p_1) \wedge f(p_2)$,
   — $f(p_1 + p_2) \rightarrow f(p_1) \vee f(p_2)$,[4]
   — $f(\sim p_1) \rightarrow \neg f(p_1)$.
3. Traduction des propriétés devant être garanties par la chaîne de fonctions de sécurité en contraintes SMTlib. Pour une chaîne de fonctions de sécurité donnée c et une propriété p du plan de données, on veut être sûr que l'implication $c \rightarrow p$ est valide. Ceci est équivalent à vérifier que $c \wedge \neg p$ n'est pas satisfiable : la dernière expression apparaît dans l'entrée de SMTlib.

Afin de illustrer ce processus, nous considérons une chaîne de sécurité simple composée de 4 fonctions de sécurité, notées F_i, avec $i \in \{1, 2, 3, 4\}$ correspondant à des pare-feux. Cependant notre solution n'est pas limitée à un seul type de fonctions de sécurité. La spécification Pyretic correspondant à cette chaîne apparaît dans le listing 1. Chaque fonction de sécurité est décrite par des compositions de règles match appliquées sur des adresses IP et des numéros de port. Par exemple, la fonction F_3 n'accepte que des paquets dont le port source est 4000, 5000 ou 6000. Ces différentes fonctions sont ensuite combinées avec les opérateurs séquentiel, parallèle et la négation afin de obtenir la chaîne finale.

```
F1 = match(srcip=IP("198.122.37.15")) +
     match(srcip=IP("253.182.3.14"))

F2 = match(srcport=1000) + match(srcport=2000) +
     match(srcport=3000)

F3 = match(srcport=4000) + match(srcport=5000) +
     match(srcport=6000)

F4 = match(dstport=7000) + match(dstport=8000) +
     match(dstport=9000)

chain = ((F1 >> F2)+(~F1 >> F3)) >> F4
```

Listing 5.1 – Spécification Pyretic d'un exemple jouet de chaîne de fonctions de sécurité.

Le comportement de la chaîne peut être décrit littéralement comme suit : si la fonction de sécurité F_1 accepte un

4. La sortie de $p_1 + p_2$ est composée de l'union de p_1 et p_2, ce qui explique la traduction de cet opérateur par une disjonction.

paquet donné, celui-ci est transmis à la fonction de sécurité F_2 ; autrement il est transmis à la fonction de sécurité F_3. Tous les paquets acceptés par F_2 ou F_3 sont finalement transmis à F_4. Supposons que nous voulions vérifier les propriétés suivantes pour la chaîne : chaque paquet accepté par F_1, F_2 et F_4 ou rejetés par F_1 et acceptés par F_3 et F_4 doivent être acceptés par la chaîne.

```
(set-option:produce-models true)
(set-logic QF_LIA)
; Declaration des variables et des valeurs
(declare-const allowed Bool)
(declare-const srcip Int)
(declare-const ip0 Int)
...
; Traduction de la chaine de fonctions de securite
(assert (and
  (distinct port3 port5 port0 port7 ip1
            port8 port1 port4 port6 port2 ip0)
  (= allowed (and
     (or (and (or (= srcip ip0) (= srcip ip1))
              (or (= srcpt port0) (= srcpt port1)
                  (= srcpt  port2)))
         (and (not(or (= srcip ip0) (= srcip ip1)))
              (or (= srcpt port3) (= srcpt port4) (= srcpt port5))))
     (or (= dstpt port6) (= dstpt port7)
         (= dstpt port8))))
; Traduction des proprietes
 (and
   (or (and (or (= srcip ip0) (= srcip ip1))
            (or (= srcpt port0) (= srcpt port1)
                (= srcpt port2))
            (or (= dstpt port6) (= dstpt port7)
                (= dstpt port8)))
       (and (not(or (= srcip ip0) (= srcip ip1)))
            (or (= srcpt port3) (= srcpt port4)
                (= srcpt port5))
            (or (= dstpt port6) (= dstpt port7)
                (= dstpt port8))))
   (not allowed))))
(check-sat)(get-model)(exit)
```

Listing 5.2 – Fragment de code SMTlib généré pour l'exemple jouet.

Le listing 5.2 montre le (très largement abrégé) résultat de la traduction de l'exemple jouet et de ses propriétés contraignantes dans le langage SMTlib, la spécification complète se trouvant à l'annexe A. Il inclut la déclaration des variables et des valeurs (abstrayant la valeur concrète des adresses IP et des numéros de ports), la traduction de la chaîne de fonctions de sécurité et la traduction (de la négation) des propriétés. En particulier, chaque opérateur de composition est traduit par un opérateur booléen, et les contraintes représentent les conditions pour l'acceptance d'un paquet par la chaîne de fonctions de sécurité. Le fichier ainsi généré est fourni à un SMT solver qui produira un verdict sur la cohérence de la chaîne, répondant que le modèle ne peut être satisfait dans le cas où la chaîne garantit les propriétés et produisant un contrexemple dans le cas contraire.

5.2.2 Algorithme de traduction vers un modèle vérifiable par model checking

Notre module de vérification supporte aussi la vérification de chaîne par application de model checking. C'est un complément direct aux possibilités apportées par Kinetic pour la vérification du plan de contrôle. Cependant la traduction d'une politique Pyretic dans un automate à états finis, pouvant être vérifié par application de model checking, est moins intuitive que la traduction dans un modèle SMTlib. Afin de effectuer cette traduction nous extrayons des sous-chaînes strictement séquentielles de la chaîne exprimée en Pyretic. La définition inductive suivante introduit la condition pour qu'une chaîne c soit strictement séquentielle :

— $C \in \{identity, drop, match, modify\}$,

— $C = \sim C_1$ où c_1 est strictement séquentielle ;
— $C = C_1 \gg C_2$ où c_1 et c_2 sont strictement séquentielles.

1. Initialement, générer seulement l'état initial et l'état final de l'automate ;
2. Créer une variable correspondant à chaque attribut des paquets (e.g. srcip, srcport) qui apparaissent dans la spécification. Ces variables seront utilisées pour spécifier des conditions de transition.
3. Pour toute composition parallèle, créer un état de l'automate. Ces états décrivent la position d'un paquet dans la chaîne de fonctions de sécurité. En particulier, un paquet est considéré comme étant accepté lorsqu'il atteint l'état final de l'automate.
4. Pour toute sous-chaîne strictement séquentielle, créer une transition de l'automate. Les conditions sur cette transition correspondent aux actions atomiques sur cette sous-chaîne.

La figure 5.2 illustre l'automate obtenu à partir de la spécification Pyretic présentée dans la sous-section précédente. Les valeurs pour chaque champ d'entête ont été abstraites par des valeurs symboliques, et les états associés à la chaîne de fonctions de sécurité sont notés $S_0, \ldots, S_5$.

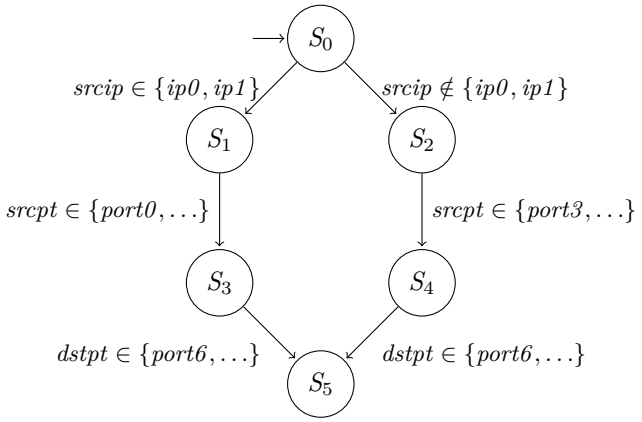

FIGURE 5.2 – Automate du plan de données pour l'exemple jouet

L'automate résultant est représenté dans un modèle nuXmv comme un automate à états finis. Il est complété par la spécification des propriétés à vérifier sur la chaîne exprimées sous la forme de conjonctions de conditions sur ses champs. Ces conditions sont traduites en utilisant la logique temporelle CTL afin de exprimer l'acceptation ou le rejet d'un paquet sous la forme d'une propriété de sûreté. Pour ce faire nous utilisons le quantificateur universel de chemins **A** et l'opérateur temporel **G** qui exprime qu'une propriété est vraie pour tout un chemin. Nous considérons en effet un paquet comme accepté s'il existe un chemin d'exécution du modèle tel qu'il atteigne finalement l'état final. Une telle propriété peut être exprimé grâce au quantificateur existentiel de chemin **E** et à l'opérateur de finalité **F**. Pour une propriété p conditionnant les champs d'entête acceptés par une chaîne nous générons donc la propriété CTL $\mathbf{A}\,\mathbf{G}(p \rightarrow (\mathbf{E}\,\mathbf{F}\,state = terminal))$ qui exprime que le paquet atteindra toujours le dernier état de la chaîne s'il vérifie bien la condition p. Nous ajoutons ainsi la traduction des propriétés associées à une chaîne à notre modèle afin de pouvoir le vérifier par le model checker nuXmv. Le listing 3 illustre les propriétés obtenues pour notre exemple, la spécification nuXmv complète se trouvant à l'annexe B.

```
AG(((srcip=ip0 | srcip=ip1) &
    (srcpt=port0 | srcpt=port1 | srcpt=port2) &
    (dstpt=port6 | dstpt=port7 | dstpt=port8))
   -> EF state=S5)

AG((((!(srcip=ip0 | srcip=ip1)) &
    (srcpt=port3 | srcpt=port4 | srcpt=port5) &
    (dstpt=port6 | dstpt=port7 | dstpt=port8))
   -> EF state=S5)
```

Listing 5.3 – Spécification formelle interprétable par un model checker pour l'exemple jouet.

5.3 Prototype et résultats expérimentaux

Nous avons évalué les performances de notre approche au travers de plusieurs expérimentations. En particulier, nous voulions comparer les performances obtenues avec les deux approches de traduction vers SMTlib et nuXmv, ainsi qu'avec différents SMT solvers afin de évaluer le surcoût introduit par Synaptic. Le contexte expérimental était basé sur un ordinateur portable MacBook Air avec un processeur Intel Core i5 (1.7 GHz), et 4Gb de mémoire RAM. Nous avons utilisé les trois outils de résolution suivants : CVC4 (version 1.4), veriT (version 201506), et nuXmv (version 1.0.1). De manière à réaliser ces expérimentations, nous avons également implémenté un module python capable de générer synthétiquement les chaînes de sécurité utilisées comme entrées de notre module de vérification. En effet, ce module ne supporte pas encore les chaînes générées par notre module de synthèse en raison de l'antériorité chronologique de nos travaux sur la vérification par rapport à nos travaux sur la synthèse. Toutefois l'application de notre approche de vérification aux chaînes spécifiées en Pyretic devrait pouvoir être rapidement atteinte, l'application à nos chaînes spécifiées en Frenetic demanderait vraisemblablement un effort plus prolongé.

Dans l'évaluation de nos travaux sur la vérification de chaînes nous prenons plusieurs paramètres en ligne de compte, en particulier nous avons considéré :

— la taille de l'automate du plan de contrôle spécifiant les changements dans la chaîne de fonctions de sécurité en réponse aux évènements réseau, mesuré en tant que nombre d'états ;
— la taille de la chaîne de sécurité exprimée en tant que longueur et largeur ;
— le nombre de propriétés qui ont besoin d'être validées pour une chaîne de sécurité.

Pour chacun de ces paramètres, nous avons évalué le temps de réponse et la consommation mémoire des outils de SMT solving et de model checking. Ces évaluations incluent la traduction de la chaîne dans un modèle formel donné et sa validation par l'outil correspondant. Nous avons utilisé le module python pour le temps et le profiler mémoire valgrind pour exécuter nos expérimentations et avons obtenus les résultats correspondants.

5.3.1 Prototypage

L'architecture de vérification ici présentée a été implémentée en Python version 2 pour un total de 3864 lignes de code, incluant les algorithmes de traduction vers les modèles formels. Les classes correspondantes sont décrites dans le diagramme 5.3. Nous avons implémenté notre module de vérification et ses algorithmes de traduction comme un package de vérification écrit en Python. Nous l'avons conçu comme une extension du langage de programmation de contrôleurs SDN Pyretic et exploité son extension Kinetic qui fait elle aussi partie de la famille de langages Frenetic. Notre module de vérification peut être utilisé pour vérifier tout à la fois la cohérence de l'automate du plan de contrôle et des règles du plan de données de chaînes de fonctions de sécurité implémentées en Pyretic. Comme nous l'avons déjà présenté dans la figure 5.1, il reçoit comme argument la spécification d'une chaîne de fonctions de sécurité implémentée en Pyretic, et retourne les résultats de vérification à l'orchestrateur, correspondant à un verdict sur la validité de la chaîne. Sur la base de cette analyse l'orchestrateur peut interagir avec le contrôleur SDN, afin de ce que la chaîne de fonctions de sécurité soit déployée ou non dans le réseau programmable.

En détail, le module de vérification repose sur deux blocs architecturaux principaux : un analyseur sémantique capable d'interpréter la chaîne et ses symboles, et un générateur de modèle capable de générer les spécifications formelles sur lesquelles s'appuient notre architecture de vérification. Le générateur de modèle est spécifié à un niveau abstrait, de telle sorte qu'il est facile de l'étendre avec de nouvelles méthodes de vérification formelle. Nous avons jusqu'à présent implémenté deux générateurs de modèle correspondant aux deux approches présentées ci-dessous :

— un générateur de modèles SMTlib capable de générer des modèles interprétables par un SMT solver tel que par exemple CVC4 et veriT,
— un générateur de modèles nuXmv produisant des modèles interprétables par le model checker nuXmv. [5]

Notre module de vérification peut premièrement vérifier le plan de contrôle associé avec une chaîne de fonctions de sécurité par application de l'extension Kinetic. Il génère ensuite la spécification formelle correspondant au plan de données et la transmet à l'outil de vérification correspondant. La vérification d'une chaîne de fonctions de sécurité donnée peut être réalisée pro-activement en explorant tous les états atteignables par l'automate de contrôle, et ensuite à toutes les configurations possibles du plan de données.

Nous avons également implémenté tout un ensemble de classes permettant la spécification de propriétés logiques devant être satisfaites par les chaînes de fonctions de sécurité. Cette architecture est décrite dans le graphique 5.4. Ce langage permet l'expression de contraintes sur les champs d'entête d'une politique au travers

5. Nous utilisons nuXmv plutôt que son prédécesseur NuSMV afin de directement exprimer les propriétés sur les attributs des paquets.

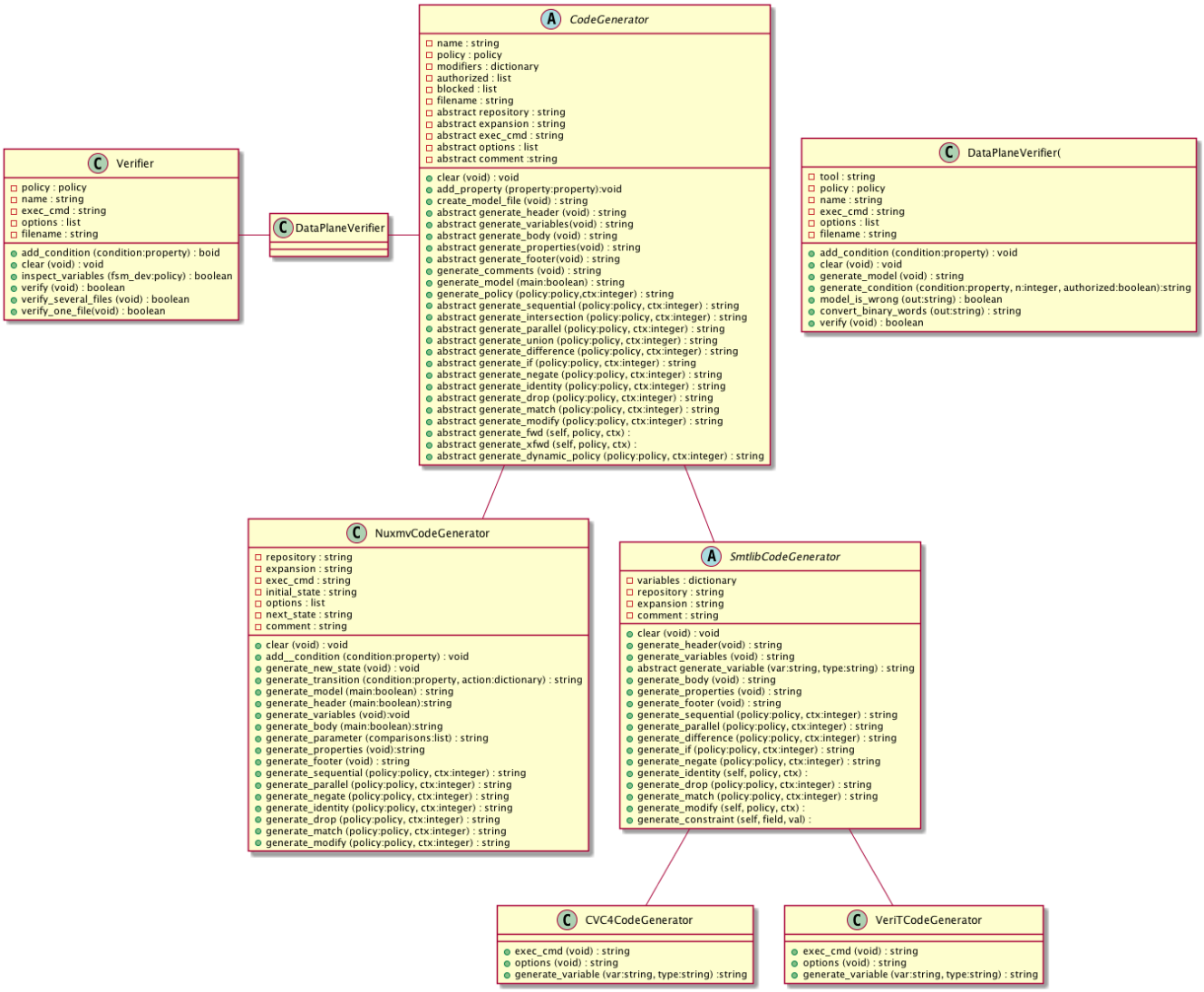

FIGURE 5.3 – Diagramme des classes de notre module de vérification de chaînes de sécurité

des différentes classes supportant des comparaisons. A partir de ces propriétés statiques il est possible de dériver des propriétés du plan de contrôle par application d'une des classes supportant des opérateurs temporels telles que par exemple AG ou EF. De telles propriétés permettent de valider que les chaînes générées pour protéger un ensemble d'applications ne contiennent pas de contradictions, dans le cas contraire celles-ci peuvent être corrigées avant le déploiement de la chaîne factorisée.

5.3.2 Impact de la taille de l'automate de contrôle sur les performances de vérification

Dans une première série d'expérimentations, nous nous sommes intéressés à quantifier l'impact de la taille de l'automate de contrôle sur les performances de vérification. Notre module de vérification supporte tout à la fois la vérification du plan de contrôle et du plan de données. La taille de l'automate de contrôle influence directement le nombre de processus de vérification requis pour vérifier la chaîne de sécurité. Considérons le cas d'une chaîne avec un automate de contrôle composé de n états. Le module de vérification doit alors exécuter $n+1$ processus de vérification, un supporté par l'extension Kinetic pour vérifier le plan de contrôle et n processus correspondant aux instances de vérification pour valider le plan de données. Nous utilisons notre module python pour générer des chaînes avec différentes tailles d'automate de contrôle, variant de 0 à 100 états afin de évaluer les performances

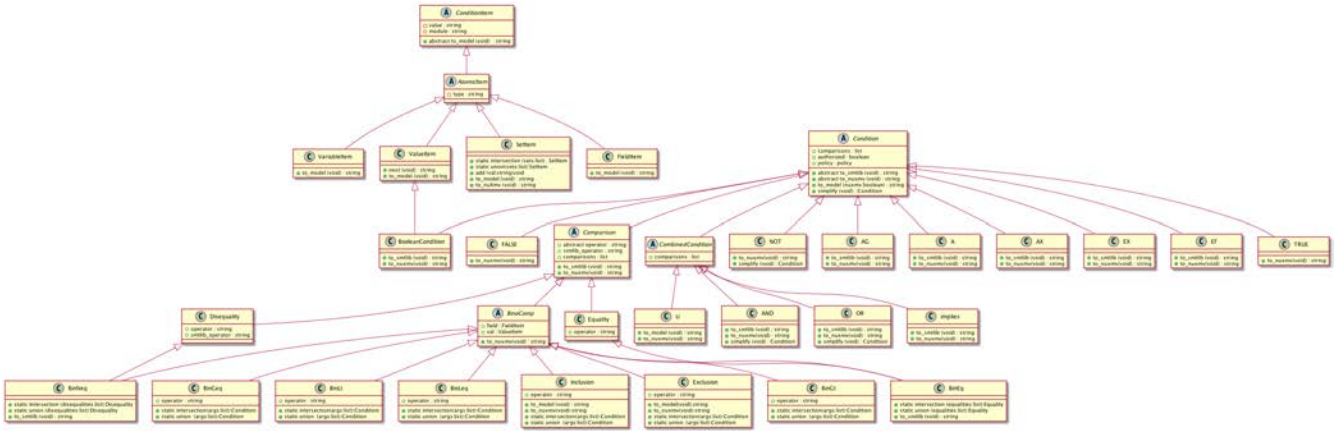

FIGURE 5.4 – Diagramme des classes de notre langage de propriétés pour les chaînes de sécurité

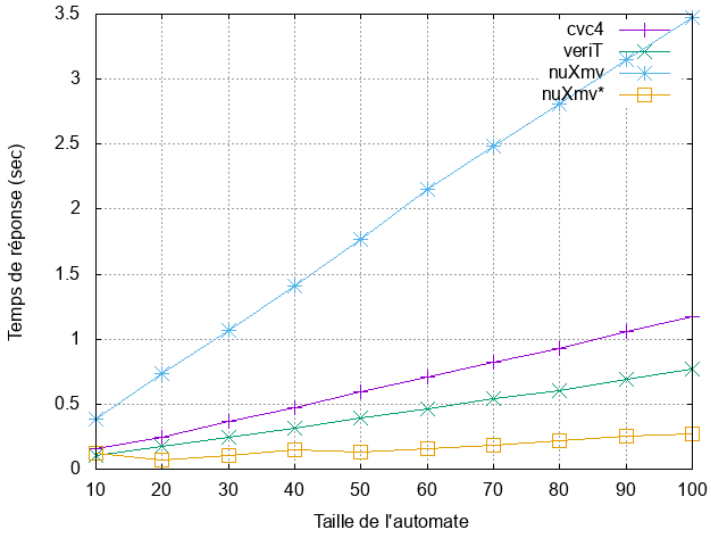

FIGURE 5.5 – Temps de réponse en fonction de la taille de l'automate de contrôle

en termes de temps de réponse et de consommation mémoire de notre module de vérification.

La figure 5.5 représente les temps de réponse (en secondes) de notre module de vérification avec différents outils de vérification : CVC4, veriT, and nuXmv. Puisque la vérification du plan de contrôle est aussi basée sur du model checking, il est possible d'exécuter la vérification des plans de données et de contrôle dans une même instance de nuXmv. Ce cas correspond à la dernière courbe notée nuXmv*. A partir de ces résultats nous pouvons observer que le temps de réponse croit linéairement en fonction de la taille de l'automate de contrôle pour chacune des méthodes de vérification. Les meilleures performances sont obtenues avec l'approche nuXmv* dans laquelle les plans de données et de contrôle sont vérifiés dans la même instance de nuXmv. De manière surprenante le pire cas est celui de l'approche basée sur nuXmv avec un processus de vérification par état de l'automate. Dans ce cas, le temps de réponse est tout de même relativement acceptable, avec un temps total de 3.5 secondes pour un automate de contrôle de 100 états.

Nous avons également quantifié la consommation mémoire avec ces différentes tailles d'automates. La figure 5.6 indique les consommation mémoires maximales obtenues pendant chaque phase de vérification. Deux courbes seulement sont visibles dans cette figure, une correspondant à l'approche nuXmv* et l'autre correspondant aux performances de CVC4, veriT et nuXmv, qui sont identiques pour ces trois dernières approches. La plus haute consommation mémoire est toujours observée pour le processus vérifiant l'automate de contrôle, basé sur l'extension Kinetic et le model checker nuXmv. Durant ces expérimentations, l'approche nuXmv* consomme davantage de mémoire tout en présentant une courbe de performance plus que linéaire lorsque l'on fait varier

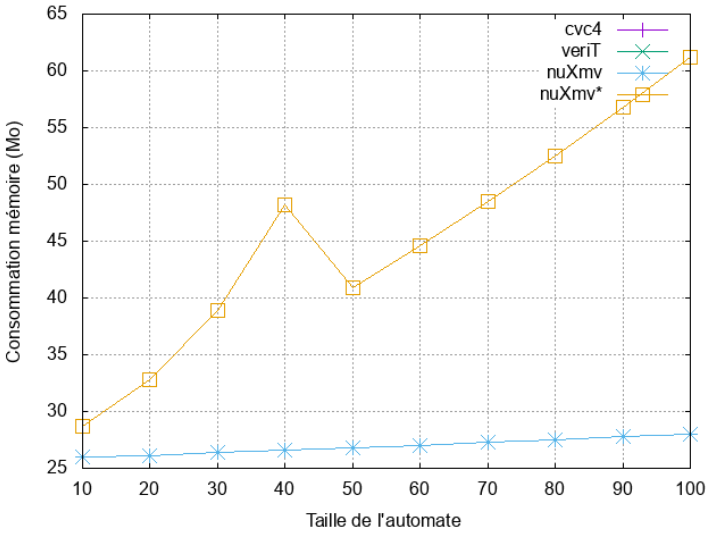

FIGURE 5.6 – Consommation mémoire en fonction de la taille de l'automate de contrôle

la taille de l'automate de contrôle. En particulier, on obtient une consommation mémoire de 62 Mb pour un automate de 100 états, le pic observé à 40 états étant vraisemblablement dû à la mise en place de stratégies de compression du modèle en mémoire. Les trois autres courbes (CVC4, veriT, nuXmv) sont caractérisées par un comportement linéaire avec approximativement un tiers de la consommation mémoire de nuXmv* pour le même automate de 100 états. Ainsi, les faibles temps de réponse fournis par nuXmv* sont contrebalancés par une forte consommation mémoire, ce qui peut constituer un facteur de limitation pour des scénarios où de nombreuses chaînes de fonctions de sécurité ont besoin d'être validées : par exemple, dans le cas de plusieurs chaînes de fonctions de sécurité protégeant plusieurs applications sur un ensemble d'équipements intelligents.

5.3.3 Impact de la largeur et de la longueur des chaînes sur les performances de vérification

Dans une seconde série d'expérimentations, nous évaluons l'impact de la largeur et de la longueur des chaînes sur notre module de vérification. Comme nous nous concentrons uniquement sur le plan de données dans ces expérimentations, nous ne distinguons pas les approches nuXmv et nuXmv*. Nous ne considérons donc que les résultats de vérification obtenus avec les vérificateurs CVC4, veriT, et nuXmv.

Nous étudions en premier lieu l'impact de la largeur des chaînes de sécurité, correspondant au nombre de fonctions (ou de règles) composées en parallèle. La figure 5.7 illustre les temps de réponse de notre prototype en faisant varier la largeur des chaînes de 100 à 1000. Les temps de réponse observés croissent bien davantage que linéairement pour les trois méthodes de vérification considérées. Nous espérions le même phénomène qu'observé lors des expériences avec les automates de contrôle. En fait les meilleures performances sont obtenues avec nuXmv. Ce dernier fournit une valeur de 0.8 secondes pour une largeur de la chaîne de sécurité de 1000, tandis que CVC4 et veriT produisent des valeurs de respectivement 6.4 et 15.9 secondes dans les mêmes conditions. Lorsque l'on analyse les résultats en termes de consommation mémoire, montrés dans la figure 5.8, on peut observer que veriT requiert 62.81 Gb de mémoire pour la chaîne la plus large, ceci pouvant être expliqué par l'algorithme de cloture des égalités utilisé par veriT qui consomme beaucoup de mémoire. CVC4 et nuXmv sont beaucoup plus efficaces et ne consomment respectivement que 29.89 Mb et 202.67 Mb dans le pire des cas.

La longueur des chaînes de sécurité correspondant au nombre de fonctions (ou règles) composées en séquence. Les figures 5.9 et 5.10 illustrent respectivement les performances en termes de temps de réponse et de consommation mémoire, en considérant une chaîne de sécurité avec une longueur variant de 10 à 100 fonctions. Ce paramètre a une influence importante sur le temps de réponse de CVC4, avec une valeur de plus de 100 secondes dans le pire des cas, tandis que les deux autres approches nuXmv et veriT, maintiennent des valeurs acceptables dans ces conditions.

Les résultats en termes de consommation mémoire sont indiquées seulement pour les vérificateurs nuXmv et veriT, étant donné que CVC4 a été disqualifié par ses mauvais temps de réponse. Il apparaît que veriT consomme moins de mémoire que nuXmv pour les plus petites chaînes de sécurité, depuis une taille de 10 jusqu'à

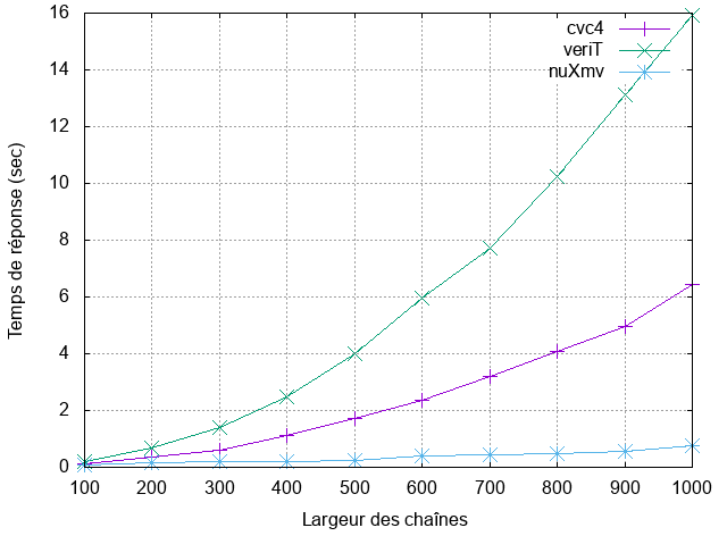

FIGURE 5.7 – Temps de réponse en fonction de la largeur de la chaîne à vérifier

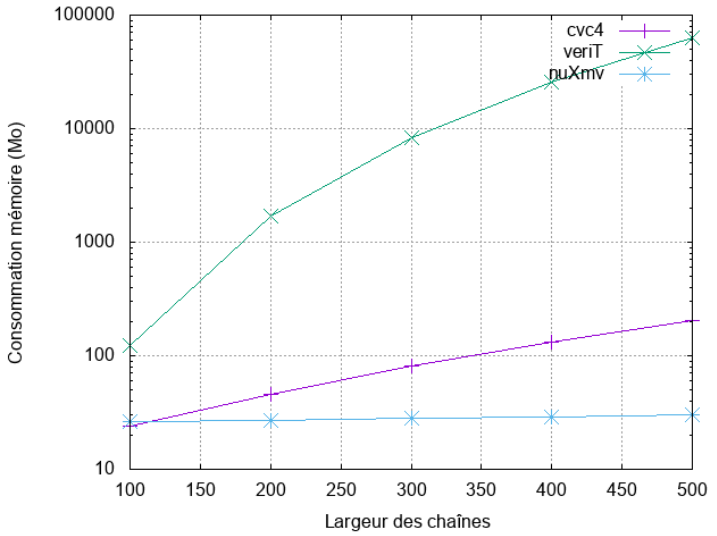

FIGURE 5.8 – Consommation mémoire en fonction de la largeur de la chaîne à vérifier

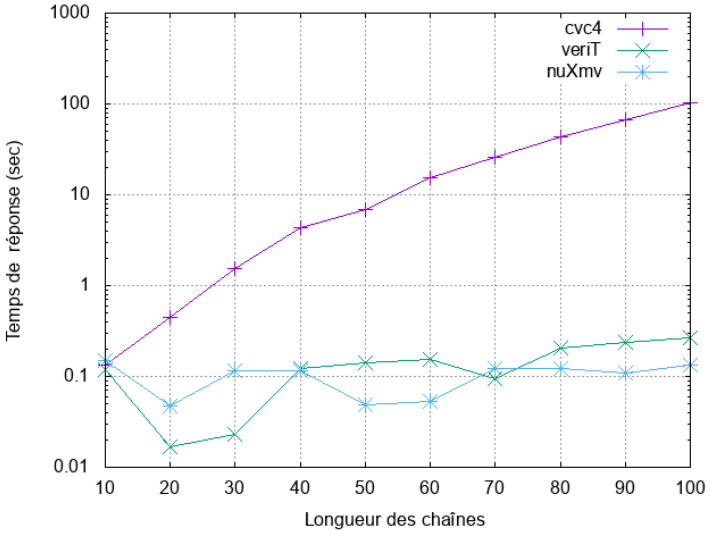

FIGURE 5.9 – Temps de réponse en fonction de la longueur de la chaîne à vérifier

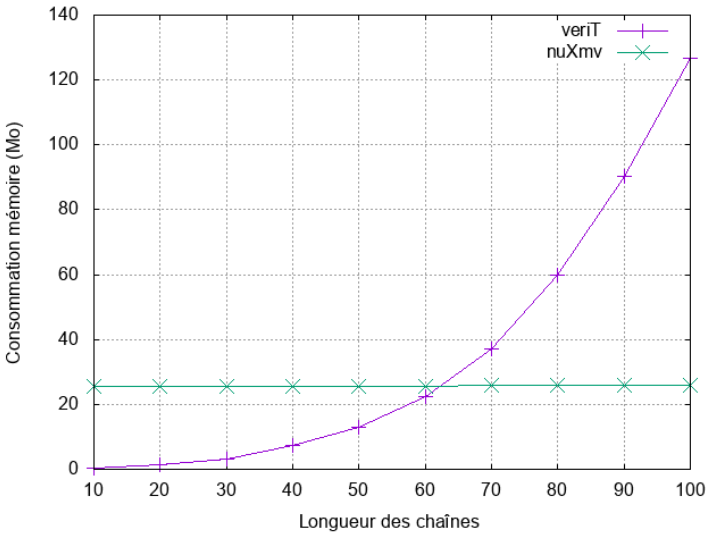

FIGURE 5.10 – Consommation mémoire en fonction de la longueur de la chaîne à vérifier

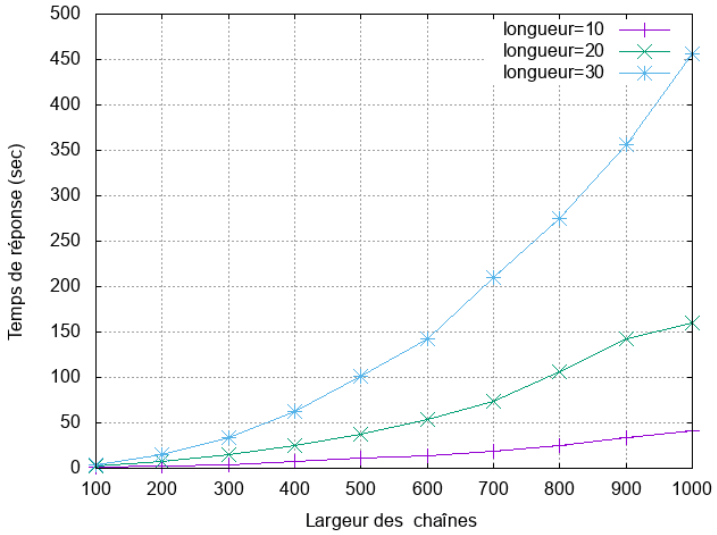

FIGURE 5.11 – Temps de réponse de nuXmv vs. à la fois la largeur et la longueur

60. Cependant, quand la longueur augmente d'environ 80 jusqu'à 100, nuXmv présente les meilleurs résultats, avec une valeur stable aux alentours de 26 Mb, tandis que veriT requiert au plus 127 Mb dans le pire cas. Ces expérimentations sur la longueur et la largeur des chaînes vont en faveur de nuXmv (model checking) au lieu de veriT et CVC4 (SMT solving), pour la vérification de chaînes de sécurité complexes.

Pour compléter ces résultats, nous avons effectué une troisième série d'expérimentations dans laquelle nous avons évalué l'influence tout à la fois de la largeur et de la longueur des chaînes avec le vérificateur nuXmv, qui apportait les meilleures performances. La figure 5.11 donne les temps de réponse que nous avons observés avec une largeur variant de 100 à 1000, et une longueur variant de 10 à 30 (correspondant aux différentes courbes dessinées), représentant un nombre total de règles de 1000 à 30000. la chaîne de fonctions de sécurité la plus complexe requiert plus de 450 secondes de temps de vérification. Certaines optimisations relatives à la représentation formelle avec nuXmv pourraient être envisagées pour réduire ces valeurs, telles que réduire le domaine associé avec les champs d'entête ou réduire le nombre d'états initiaux en se basant sur les propriétés de sécurité qui doivent être vérifiées.

Bien que ces expérimentations aient été réalisées sur des chaînes générées synthétiquement elles donnent un ordre de grandeur des résultats que l'on obtiendrait pour des chaînes réelles. En effet le modèle de chaînes considéré ici pourrait être utilisé pour la vérification formelle de chaînes de fonctions de sécurité qui seraient alors traduites depuis leur modèle en Pyretic ou en Frenetic, nous obtiendrions alors des modèles équivalents à ceux présentés dans cette section.

5.3.4 Impact du nombre de propriétés à vérifier sur les performances de vérification

Dans une dernière série d'expérimentations, nous étudions plus spécifiquement l'influence du nombre de propriétés devant être garanties par la chaîne de sécurité. A nouveau du fait des performances précédemment observées nous ne considérons que le model checker nuXmv, pour lequel les propriétés sont exprimées avec la logique temporelle CTL.

La figure 5.12 montre à la fois les performances en termes de temps de réponse et de consommation mémoire garanties par notre module de vérification avec le vérificateur nuXmv, en considérant une chaîne de sécurité composée de 1000 fonctions, et en faisant varier le nombre de propriétés à vérifier de 100 à 1000. Les performances semblent être linéaires du nombre de propriétés à vérifier. Plus précisément, le temps de réponse varie de 0.69 à 1.46 secondes, tandis que la consommation mémoire varie de 43.34 à 173.99 Mb.

Ces performances pourraient être améliorées en fonction de la nature des propriétés. Par exemple lorsque l'on vérifie de simples invariants au lieu des propriétés temporelles complexes, nous pourrions utiliser des algorithmes plus efficaces mis à disposition par nuXmv.

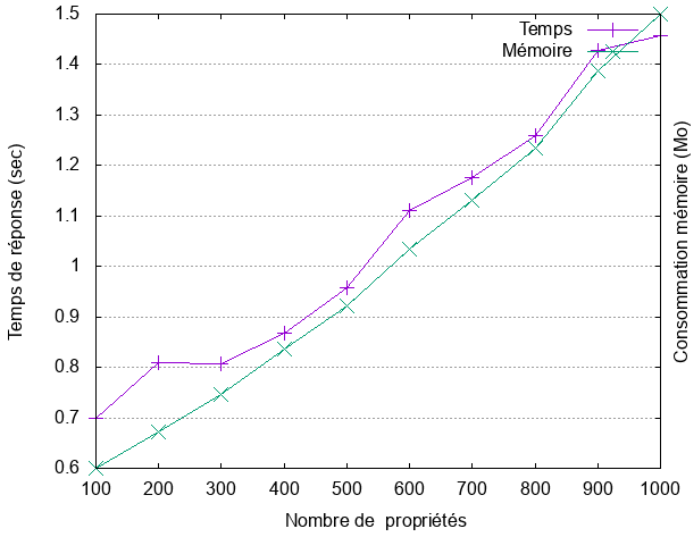

FIGURE 5.12 – Performance de nuXmv vs. nombre de propriétés devant être validées

5.4 Résumé

Dans ce chapitre nous avons présenté les résultats de nos travaux relatifs à la vérification formelle de chaînes de fonctions de sécurité spécifiées en Pyretic. Nous nous sommes appuyés sur une stratégie de vérification couvrant tout à la fois le plan de contrôle et le plan de données d'une chaîne, pour ce faire nous avons étendu la procédure de vérification formelle proposée par Kinetic avec une stratégie de vérification pour le plan de données. Cette stratégie s'appuie sur la traduction automatique des chaînes en SMTlib ou dans le langage d'entrée de nuXmv afin d'en assurer la vérification par les outils dédiés. Nous avons évalué les performances apportées par différents outils de vérification pour chacun de ces formalismes. Au terme de ces évaluations, nous avons identifié nuXmv comme étant le meilleur candidat pour implanter la vérification des chaînes en temps réel.

Les travaux présentés dans ce chapitre souffrent néanmoins de deux limites majeures. Premièrement, le langage de spécification de propriétés que nous avons développé avec Synaptic ne permet pas présentement la prise en compte des besoins particuliers des utilisateurs dérivés des automates markoviens présentés au chapitre 3, nous en avons présenté le modèle exploité par notre orchestrateur dans le chapitre 4. En outre, les performances que nous avons observées pour la vérification de nombreuses propriétés limitent encore l'utilisation de méthodes de vérification formelle pour assurer une telle validation sur de larges chaînes en temps réel. Ce constat pourrait toutefois être contourné par une refonte du modèle SMTlib de nos chaînes : en effet, après avoir expérimenté une modélisation basée sur des mots binaires et plus sur des constantes symboliques, nous avons constaté une très nette amélioration des performances apportées par CVC4 et veriT.

Les travaux présentés dans ce chapitre et dans le précédent permettent la vérification de certaines propriétés des chaînes, toutefois leur déploiement tel quel dans le réseau conduirait au placement de chaque règle sur tous les switches du réseau. Il est donc encore nécessaire de procéder au placement des règles constitutives d'une chaîne dans le réseau, ce point fera l'objet du prochain chapitre.

6

Placement et déploiement de chaînes de fonctions de sécurité

Dans le chapitre précédent, nous avons présenté comment les chaînes de fonctions de sécurité peuvent être rassemblées au travers d'un algorithme de factorisation. Toutefois l'idée de la concaténation de deux fonctions de sécurité en une seule lève intuitivement la question de la décomposition d'une unique fonction en deux. Au point de vue fonctionnel, il n'y a pas de raison d'avoir recours à une telle décomposition, en revanche celle-ci peut être motivée par des soucis d'optimisation des ressources du réseau ou bien par les contraintes matérielles qu'il induit. Pour des raisons de disponibilité certaines fonctions peuvent encore être répliquées en plusieurs points du réseau. Ces deux types d'opération, décomposition et réplication, doivent être planifiées de sorte à optimiser l'utilisation des ressources du réseau ainsi que le taux de disponibilité de la chaîne. L'établissement d'un plan de déploiement optimal pour une chaîne sur un réseau donné doit donc être traitée par l'utilisation d'outils de résolution formelle dédiés.

Dans ce chapitre, nous traitons les questionnements relatifs au placement d'une chaîne de fonctions de sécurité dans un réseau SDN. C'est dans le cadre de ce placement motivé par des questions de performances que peuvent intervenir les opérations de décomposition et de réplication. Nous commençons par présenter le problème du placement d'une chaîne de fonctions de sécurité en langage naturel. Ce problème pouvant mobiliser d'importantes ressources de calcul nous en donnons ensuite deux optimisations avant d'en définir précisément les paramètres et les variables, après quoi nous en définissons les contraintes et les objectifs au nombre de trois dans notre contexte : l'utilisation du réseau, la congestion du service et enfin la probabilité de sa disponibilité. Pour terminer, nous présentons l'évaluation des performances obtenues en comparant trois types de méthodes d'optimisation, à savoir la méthode linéaire de simplex solving et les méthodes non linéaires de MINLP solving et de SMT solving optimisant.

6.1 Modèle de placement

Nous détaillons tout d'abord le modèle de placement considéré pour notre approche.

6.1.1 Motivation du problème

Placer une chaîne de fonctions de sécurité sur un réseau sous-jacent ne peut se faire sans satisfaire certaines contraintes. Celles-ci peuvent provenir de la chaîne elle-même : le placement des fonctions de sécurité doit respecter l'ordre imposé par la chaîne. Les règles utilisées pour implémenter les fonctions de sécurité doivent être physiquement rattachées à des switches du réseau, ces switches ont une capacité limitée en termes de règles et sont interconnectés par des canaux de communication. Ces canaux ont eux-mêmes leur propre capacité en termes de charge de trafic. Dans un tel environnement contraint, les opérateurs réseau sont intéressés par la minimisation de certaines métriques telles que le nombre de switches utilisés dans le réseau, la congestion du service ou encore sa probabilité de disponibilité.

Pour donner un simple exemple, imaginons une chaîne avec 5 fonctions de sécurité $\{f_1, \ldots, f_5\}$. Cette chaîne surveille le trafic vers 100 destinations $\{d_1, \ldots, d_{100}\}$ ayant respectivement une charge de 10, donc nous avons un total de $5 \times 100 = 500$ règles à placer. Ces règles doivent être placées sur un réseau contenant 11 switches $\{S_1, \ldots, S_{11}\}$ connectés en séquence ou en parallèle comme présenté sur la figure 6.1 qui décrit une topologie

illustrative pour l'application de notre méthode. La capacité d'accueil de règles de chaque switch est 100 sauf pour S_3 et S_4, qui ont une capacité d'accueil de règles de 50, et une probabilité de disponibilité de 0.999 ; la capacité d'accueil de trafic de chaque canal est présentée en figure 6.1. Nous cherchons à placer les règles des 5 fonctions de sécurité sur les switches de sorte à ce que le trafic analysé par une fonction de sécurité puisse être transmis à la fonction suivante au travers d'un des canaux du réseau. Nous ne pouvons pas dépasser la capacité d'accueil de règles de chaque switch ou la capacité de transmission de charge de chaque canal.

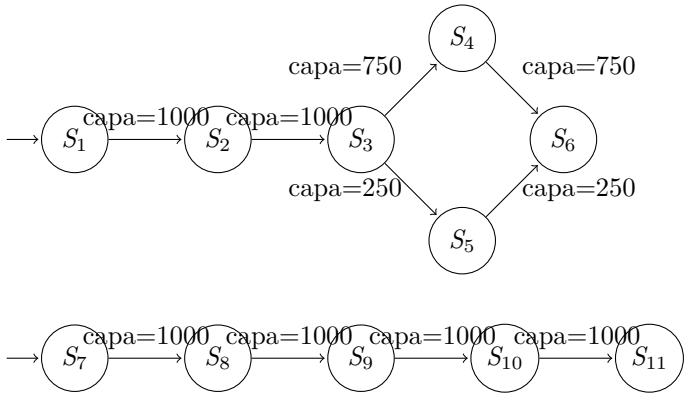

FIGURE 6.1 – Topologie réseau pour l'exemple de placement

Plusieurs solutions de placement peuvent être envisagées dans cette topologie. En premier lieu, il serait possible de placer toutes les fonctions de sécurité en séquence de S_7 à S_{11} sans introduire ni décomposition ni réplication. Cette solution minimise le nombre de switches utilisés, en revanche elle maximise la congestion du service et minimise son taux de disponibilité.

Il est encore possible de placer les trois premières fonctions de la chaîne sur S_1, S_2 et S_3 avant de décomposer la quatrième sur S_4 et S_5 avant de placer la dernière sur S_6. Cette solution introduit une décomposition due à l'impossibilité de placer l'intégralité des règles de la fonction 4 sur S_4 ou sur S_5.

Ces deux premières solutions posent le problème de saturer les switches qu'elles utilisent, il serait possible d'atteindre une troisième solution qui éviterait ce désavantage en utilisant tous les switches de la topologie mais en décomposant la chaîne de sorte à placer une moitié des règles de S_1 à S_6 d'une part et de S_7 à S_{11} d'autre part. Cette solution optimise la congestion du service, en revanche son taux de disponibilité reste inchangé.

Pour résoudre ce problème, il est possible de reprendre la précédente solution en répliquant toutes les règles de la chaîne sur les deux sous-ensembles de switches, on maximise ainsi le taux de disponibilité au détriment de l'utilisation du réseau et de la congestion du service.

On peut également imaginer le cas où une chaîne de fonctions de sécurité ne peut pas être placée sur une topologie réseau donnée. Dans ce cas, on ne veut pas simplement optimiser le placement des règles, on veut avoir une idée de comment modifier le réseau de sorte à pouvoir placer la chaîne. Une telle approche est appelée provisionnement (ou *provisioning*) de ressources. Dans un contexte SDN, il est possible de modifier le réseau en fonction du résultat du provisionnement de ressources avant de ré-optimiser la chaîne sur le réseau mis à jour.

6.1.2 Stratégies d'abstraction du modèle de placement

Dans de réels problèmes de placement, on peut avoir plus de 100 destinations à surveiller et plus de 11 switches dans le réseau. Nos prochains cas d'étude mettent en jeu des centaines de destinations à surveiller ainsi que de multiples switches, et calculer l'ensemble exact des règles placées sur chaque switch n'est pas faisable en pratique. Pour minimiser la complexité du problème général, nous introduisons ici deux abstractions du modèle, une concernant les destinations à surveiller et une concernant les switches sur lesquels les les fonctions doivent être placées placer.

Les destinations peuvent être agrégées de sorte à réduire leur nombre total. Dans notre contexte, les règles surveillant une adresse IP portent d'ores et déjà une certaine sémantique et l'on n'a pas besoin de savoir précisément où elles sont placées dans le réseau. On peut considérer des agrégats d'adresses IP qui peuvent être associés avec les switches et ensuite placés sur les équipements correspondants. Nous appelons ces collections d'adresses IP *agrégats de destinations*. Il est important pour l'optimalité du problème de placement que les agrégats d'adresses IP représentent des charges de trafic comparables. Pour obtenir un tel résultat, nous calculons les agrégats d'un problème de placement comme les résultats d'un problème d'optimisation de sac à dos. Le nombre de sacs à

Chemin	Origine	Longueur	Capacité (règles)	Capacité (charge)	Disponibilité
p_1	s_1	3	100	1000	0.997
p_2	s_4	1	50	750	0.999
p_3	s_5	1	50	250	0.999
p_4	s_6	1	100	1000	0.999
p_5	s_7	5	100	1000	0.995

FIGURE 6.2 – Caractéristiques des chemins de la topologie exemple

dos est calculé comme le ratio entre la charge totale à placer et la capacité du plus petit canal dans le réseau de sorte à garantir que chaque agrégat puisse être placé sur chaque canal. Dans notre exemple, nous avons 100 destinations avec chacune une charge de 10, donc une charge totale de 1000. La capacité minimale d'un canal est 250, donc finalement nous allons considérer des agrégats de destinations $\{D_1, D_2, D_3, D_4\}$. Chacun de ces agrégats correspond à une charge de 250 et à un poids de 25 correspondant au nombre de destinations IP agrégées.

Les switches peuvent être agrégés comme des chemins réseau : un chemin est une succession de n switches connectés en séquence sans alternative de branchement. Lorsque le trafic entre dans un tel chemin, il sera alors analysé par tous les switches sans interruption, on a alors seulement besoin de considérer le placement des règles sur le premier switch du chemin. Le placement des règles sur un chemin particulier n'induit aucune modification du traitement qui leur est associé, en fait un tel placement est comparable à l'optimisation de l'utilisation des registres d'un processeur dans le cas d'un algorithme de compilation. Les propriétés des chemins réseau sont calculées en fonction de leurs switches et canaux internes. Nous considèrerons les propriétés suivantes dans la suite de ce chapitre :

— *length* : le nombre de switches connectés en séquence ;
— *rule_capacity* : la capacité d'accueil de règles minimale dans le chemin ;
— *load_capacity* : la capacité de transfert de charge minimale dans le chemin ;
— *path_probability* : la propriété de disponibilité du chemin.

Dans notre exemple nous avons 5 chemins réseau dont les propriétés sont résumées sur la figure 6.2. Signalons enfin que ces deux abstractions n'altèrent en rien l'optimalité de la solution du problème de placement. L'abstraction sur les chemins n'induit pas de modification structurelle des conditions de placement d'une chaîne, il suffit donc de garantir que l'intégralité de la topologie est fournie en entrée du problème de placement pour en garantir l'optimalité. Pour l'abstraction sur les destinations, celle-ci entraîne une perte de granularité de la répartition de la charge de trafic générée par une application, cette perte étant toutefois négligeable tant que le modèle préserve bien les conditions de satisfiabilité du problème de placement.

6.1.3 Variables du problème de placement de chaînes de fonctions de sécurité

L'information décrivant les chaînes et le réseau sont fournies comme paramètres à notre problème de placement. Les agrégats de destinations sont représentés par un ensemble *dests*. Chaque (agrégat de) destination(s) implique un certain nombre de flux à gérer, nous représentons cette charge de trafic par un dictionnaire *dest_loads* indexé par l'ensemble *dests*. De manière similaire, nous associons à chaque agrégat d'adresses IP un poids représentant son nombre de destinations agrégées, nous notons ce dictionnaire *dest_weights*. Dans notre exemple, chaque agrégat de destinations a un poids de 25 correspondant à son nombre d'adresses IP, cette information est ensuite fournie avec l'information sur la charge du trafic.

Nous définissons le poids des fonctions de sécurité comme étant leur nombre de règles par destination. Pour chaque fonction de sécurité, notre algorithme de génération garantit que chaque destination sera protégée par exactement deux règles, une pour le trafic entrant et une pour le trafic sortant. Cette information traitant le poids des fonctions de sécurité est gérée par un dictionnaire additionnel *function_weight* qui sera aussi utilisé pour calculer le nombre de règles gérées par une fonction de sécurité.

Les chemins réseau sont représentés par un ensemble *chemins* correspondant à la première colonne du tableau 6.2. A partir de cet ensemble, nous initialisons plusieurs dictionnaires : *path_lengths* donne l'information à propos de la longueur de chaque chemin, *rule_capacities* associe la capacité d'accueil de règles minimale à chaque chemin, *load_capacities* stocke la capacité des chemins en termes de charge et *path_probability* indique la probabilité de disponibilité de chaque chemin. Le dictionnaire à deux dimensions *path_connections* indique si un chemin est le successeur d'un autre chemin. Finalement, nous dérivons deux ensembles *incomings* et *outgoings* qui représentent les chemins entrants et sortants du réseau. En détails, $i \in incomings$ si et seulement si $\forall pt \in chemins,\ path\_connections_{(pt,i)} = 0$ et $o \in outgoings$ si et seulement si $\forall pt \in chemins,\ path\_connections_{(o,pt)} = 0$.

Les variables pour représenter le placement des règles sont *dest_placement*, une matrice de variables binaires indexée par *chemins* et *dests* qui représente le fait que les règles d'une chaîne surveillant un agrégat de destinations d sont placées sur un chemin p, et le tableau *used_chemins* de variables binaires indexé par *chemins* pour identifier les chemins réseau qui sont utilisés.

Pour illustrer les gains introduits par nos abstractions, considérons la variable *dest_placement*. sans abstraction nous aurions 500 règles à placer et 11 switches, résultant à un total de $500 \times 11 = 5500$ variables binaires. Avec nos abstractions, nous avons simplement 4 agrégats à considérer ainsi que 5 chemins réseau, résultant dans un total de 20 variables binaires. Le gain le plus significatif est observé pour le nombre de règles qui est abstrait à un nombre d'agrégats d'adresses IP. Au lieu de considérer le nombre de règles pour chaque destination IP pour chaque fonction de sécurité, nous n'avons besoin de considérer que l'agrégat IP à placer pour toute la chaîne de fonctions de sécurité modulo la longueur des chemins réseau.

6.1.4 Modèle des contraintes

Notre procédure d'optimisation est sujette aux contraintes suivantes qui représentent les conditions pour un arrangement pour être un placement de règles valable. Nous indiquons également comment certaines contraintes doivent être relaxées pour satisfaire à un problème de provisionnement de ressources. Le premier ensemble de contraintes que l'on considère caractérise les chemins utilisés dans le réseau. Par souci de continuité avec les contraintes suivantes nous avons fait le choix d'exprimer ces contraintes sous forme arithmétique en traduisant la valeur booléenne "VRAI" par l'entier 1 et la valeur booléenne "FAUX" par l'entier 0.

1. Un chemin est utilisé si les règles pour au moins un agrégat de destinations sont placées dessus :
   $\forall p \in chemins,\ |dests| \times used\_chemins_p \geq \sum_{d \in dests} dest\_placement_{(p,d)}$
2. Un chemin peut être utilisé seulement si au moins un de ses successeurs est utilisé :
   $\forall p \in chemins,\ |dests| \times used\_chemins_p \ \geq \sum_{suc \in chemins} path\_connections_{(p,suc)} \times used\_chemins_{suc}$
3. Un chemin peut être utilisé seulement si au moins un de ces prédécesseurs est utilisé :
   $\forall p \in chemins,\ |dests| \times used\_chemins_p \ \geq \sum_{pred \in chemins} path\_connections_{(pred,p)} \times used\_chemins_{pred}$

Le prochain groupe de contraintes caractérise le placement des agrégats de règles dans le réseau : chaque destination doit être supportée depuis l'entrée du réseau jusqu'à ses points de sortie, alors que la cohérence du placement est garantie par le précédent ensemble de contraintes que nous avons définies.

1. Chaque destination doit être placée sur au moins un chemin entrant :
   $\forall dn \in dests,\ \sum_{p \in incomings} dest\_placement_{(p,dn)} \geq 1$
2. Chaque destination doit être placée sur au moins un chemin sortant :
   $\forall dn \in dests,\ \sum_{pt \in outgoings} dest\_placement_{(p,dn)} \geq 1$

Dans le cas où la charge totale générée par les différentes destinations à surveiller ne peut pas être gérée par le réseau, ces contraintes ne peuvent pas être satisfaites, alors nous pouvons les substituer par les contraintes permettant de gérer le problème converse de provisioning de ressources.

1. Chaque destination doit être placée sur au plus un chemin entrant :
   $\forall dn \in dests,\ \sum_{p \in incomings} dest\_placement_{(p,dn)} \leq 1$
2. Chaque destination doit être placée sur au plus un chemin sortant :
   $\forall dn \in dests,\ \sum_{pt \in outgoings} dest\_placement_{(p,dn)} \leq 1$

Le dernier groupe de contraintes caractérise les capacités du réseau en termes de charge de trafic, de capacités d'accueil de règles et de charge maximale. Ces contraintes dépendent des propriétés calculées pour chaque chemin : sans cette abstraction, le nombre de contraintes sur la capacité de chaque canal serait quadratique en fonction du nombre de switches alors que dans notre modèle il est linéaire en fonction du nombre de chemins.

1. Contraintes sur la capacité d'accueil de règles de chaque chemin dans le réseau :
   $\forall p \in chemins,\ rule\_capacities_p \ \geq function\_weight \times \sum_{dn \in dests} dest\_weights_{dn} \times dest\_placement_{(p,dn)}$
2. Contraintes sur la charge de trafic de chaque chemin dans le réseau :
   $\forall p \in chemins,\ load\_capacities_p \ \geq \sum_{dn \in dests} dest\_loads_{dn} \times dest\_placement_{(p,dn)}$

6.1.5 Objectifs d'optimisation

Sous l'ensemble de contraintes introduites précédemment, nous souhaitons optimiser plusieurs objectifs. Les trois critères de notre procédure d'optimisation du placement de chaînes sont (i) l'utilisation du réseau, i.e. le nombre de switches soutenant la chaîne de fonctions de sécurité, (ii) la congestion du service, résultant d'une concentration de la charge du trafic sur quelques canaux et (iii) le risque de disponibilité, i.e. la probabilité pour le service d'être rendu indisponible du fait de pannes du réseau. Dans notre cas, ces trois critères sont combinés dans une unique fonction objectif à minimiser.

Basé sur l'abstraction des chemins que nous avons introduite pour minimiser la complexité du problème, l'utilisation du réseau peut être exprimée comme la somme des longueurs des différents chemins utilisés dans le réseau, plus formellement $\sum_{p \in chemins} path\_lengths_p \times used\_chemins_p$.

La congestion du service est définie comme le nombre de flux transporté par chaque chemin du réseau. Une première solution intuitive pour minimiser cet objectif serait de minimiser la charge de chaque chemin réseau, cependant nous pouvons simplifier cette expression en observant que minimiser la charge transmise au travers de chaque chemin du réseau est équivalent à minimiser la charge maximale transmise dans le réseau. Pour calculer ce maximum, nous introduisons la variable *max_load*, calculée en ajoutant les contraintes suivantes à notre modèle :

$\forall p \in chemins,\ max\_load \geq \sum_{dn \in dests} dest\_loads_{dn} \times dest\_placement_{(p,dn)}$

Notre troisième critère d'optimisation est la probabilité de disponibilité du service. Celle-ci est calculée à partir de la probabilité de chaque switch d'être disponible. Une caractérisation plus fine de cette probabilité devrait théoriquement introduire la probabilité de disponibilité de chaque canal, toutefois pour minimiser la complexité de notre formulation nous considérons cette dernière comme intégrée au calcul de la probabilité d'un chemin. Nous considérons les différentes probabilités de notre système comme étant indépendantes. La probabilité pour chaque chemin d'être disponible est calculée à partir de son point d'entrée : nous introduisons un nouveau paramètre *path_probs* indexé par *chemins* qui contient la probabilité d'être disponible pour chaque chemin. La probabilité de disponibilité totale est calculée à partir des valeurs de *used_chemins*. Concernant le calcul de probabilités d'unions lorsque plusieurs chemins sont utilisés en parallèle nous utilisons la borne inférieure décrite dans [66]. Pour un ensemble de chemins parallèles noté *parallel_chemins*, nous définissons la fonction *union_prob* comme suit :

$$\begin{aligned} &union\_prob(parallel\_chemins) = \\ &\quad \left(\sum_{p \in parallel\_chemins} path\_probs_p\right) - \\ &\quad \left(\sum_{p_1 \in parallel\_chemins} \sum_{p_2 \in parallel\_chemins} used\_chemins_{p_1} \times path\_probs_{p_1} \times \right. \\ &\qquad \left. used\_chemins_{p_2} \times path\_probs_{p_2}\right) \end{aligned}$$

Considérons à nouveau notre exemple. Nous notons :

$$\begin{aligned} P' = &used\_chemins_{p_1} \times path\_probs_{p_1} \times union\_prob(\{p_2, p_3\}) \times \\ &used\_chemins_{p_4} \times path\_probs_{p_4} \end{aligned}$$

la probabilité de disponibilité du sous-réseau commençant avec p_1. Une borne inférieure sur la probabilité de disponibilité globale du réseau est alors :

$$P = P' + used\_chemins_{p_5} \times path\_probs_{p_5} - P' \times used\_chemins_{p_5} \times path\_probs_{p_5}$$

Cette formule exprime la probabilité de la disponibilité de la chaîne dans le cas où cette dernière est répliquée sur les deux groupes de switches de S_1 à S_6 (P') d'une part et de S_7 à S_{11} d'autre part. Elle implique le calcul de la probabilité de l'union des disponibilités de ces deux groupes, cette dernière impliquant le calcul de l'intersection de ces deux évènements qui est ainsi retranchée à la somme de leurs probabilités respectives.

Pour maximiser la probabilité de disponibilité globale du service, nous avons alors simplement à minimiser l'opposé de P, que nous notons $-P$, de sorte à avoir notre dernier objectif d'optimisation.

Evidemment, il n'est pas possible d'exprimer le produit de variables binaires du produit ci-dessus avec des solvers linéaires. Dans ce contexte, nous procédons à une simulation exacte d'un tel produit grâce à une matrice de variables binaires *intersection_probs* indexée par *chemins*. La valeur de ces variables binaires est calculée en ajoutant les contraintes suivantes à notre modèle de placement.

1. $\forall p_1 \in chemins, p_2 \in chemins.\ intersection\_probs_{(p_1,p_2)} \leq path\_probs_{p_1}$
2. $\forall p_1 \in chemins, p_2 \in chemins.\ intersection\_probs_{(p_1,p_2)} \leq path\_probs_{p_2}$
3. $\forall p_1 \in chemins, p_2 \in chemins.\ intersection\_probs_{(p_1,p_2)} \geq path\_probs_{p_1} + path\_probs_{p_2} - 1$

Nos trois critères d'optimisation sont antagonistes : minimiser l'utilisation du réseau conduit à grouper les règles sur quelques switches et donc à maximiser la congestion du service et sa probabilité de disponibilité. Pour gérer ce problème nous calculons le front de Pareto représentant l'ensemble des solutions admissibles, L'optimum

est finalement sélectionné en fonction de priorités définies par l'opérateur du réseau. D'autres stratégies d'évaluation peuvent être imaginées, par exemple une évaluation uniforme ou une évaluation pondérée des objectifs d'optimisation.

Pour calculer la sommation des différents objectifs, nous avons besoin de standardiser ces derniers sur $[0; 1]$ de sorte à les rendre comparables. Afin qu'il en soit ainsi, nous divisons l'utilisation du réseau par la somme des longueurs des chemins du réseau $\sum_{p \in chemins} path\_lengths_p$, la congestion du service par la somme des charges de chaque destination $\sum_{d \in dests} dest\_loads_d$, et la probabilité de la disponibilité du service par une borne supérieure obtenue en supprimant toutes les variables binaires de son expression, ce qui revient à supposer que toutes ces variables soient vraies.

Dans le cas où nous exécutons le modèle pour un problème de provisioning de ressources, alors l'objectif devient un problème de maximisation. Dans ce cas nous essayons de placer autant de règles que possible sur le réseau et l'on compte combien ont pu effectivement être placées. Ce résultat sera ensuite utilisé pour mettre à jour la topologie du réseau : dans un contexte SDN il est possible d'allouer dynamiquement de nouveaux switches et canaux de sorte à gérer la charge additionnelle en redirigeant l'excédent de trafic vers les chemins ainsi nouvellement créés.

Pour illustrer ceci avec notre exemple, supposons que l'on souhaite minimiser l'utilisation du réseau, alors une solution possible serait de placer les 4 agrégats de destinations sur p_5. Dans ce cas nous n'utiliserions que 5 switches mais avec une congestion du service de 1000 et une probabilité de disponibilité de 0.995. Si nous choisissons de n'optimiser que la congestion du service alors nous utiliserions tous les switches du réseau avec une congestion de 500 correspondant à la division des agrégats de destinations en deux lots, un placé sur p_1, p_2, p_3 et p_4 avec une probabilité de disponibilité de 0.996 et un placé sur p_5 avec une probabilité de disponibilité de 0.995, donc finalement la disponibilité du service serait de 0.995. Si nous choisissons d'optimiser la disponibilité du service, nous utiliserions de nouveau tous les switches du réseau mais cette fois-ci avec une congestion du service de 1000 correspondant à la réplication de tous les agrégats de destinations sur tous les chemins du réseau. Dans ce contexte, nous obtenons une probabilité de disponibilité du service de 0.99998.

6.2 Intégration du modèle de placement dans l'architecture d'orchestration

6.2.1 Architecture du module d'optimisation

L'interaction des différents modules intervenant dans le processus d'optimisation de chaînes de fonctions de sécurité est présentée dans la figure 6.3. Outre l'orchestrateur et le module d'optimisation de chaînes dont il a déjà été question dans le chapitre 3, la figure présente un gestionnaire d'optimisation dont le rôle est d'offrir une interface générique indépendante du solver utilisé pour exécuter le processus d'optimisation. Ce diagramme présuppose que les différentes applications à protéger ont été traitées, que les chaînes correspondantes ont été générées et factorisées et enfin que les informations relatives au trafic correspondant ont été ajoutées au module d'optimisation de chaîne.

La responsabilité assignée au module d'optimisation de chaînes est ici d'assurer le calcul des paramètres du modèle d'optimisation décrit dans ce chapitre. Pour ce faire, il commence par charger les informations relatives au réseau depuis un fichier de configuration dédié puis il calcule le poids de chaque fonction dans la chaîne ainsi que la charge induite pour chaque destination. Ces différentes informations sont ensuite transmises au gestionnaire d'optimisation. Ce dernier commence par calculer la valeur des autres paramètres du modèle d'optimisation, essentiellement les chemins, leur longueur, leurs capacités en termes de règles et de flux et leur probabilité de disponibilité.

A partir de ces différentes informations, le gestionnaire d'optimisation génère ensuite le modèle correspondant au problème de placement de la chaîne de fonctions de sécurité dans le réseau et tente de le résoudre. En cas d'échec à placer la chaîne dans le réseau le module d'optimisation de chaînes demande alors la résolution d'un problème de provisioning. Le gestionnaire d'optimisation est alors chargé de générer le modèle correspondant et de le résoudre avant de transmettre les directives de mise à jour du réseau au module d'optimisation de chaînes. Dans tous les cas au terme de la résolution du problème de placement ou de provisioning le module d'optimisation de chaîne utilise la solution calculée par le gestionnaire d'optimisation pour associer un emplacement dans le réseau à chaque règle de la chaîne de fonctions de sécurité.

Une fois ce processus de placement terminé le module d'optimisation de chaînes rend la main à l'orchestrateur qui se charge alors du déploiement. S'il est nécessaire de procéder à la mise à jour du réseau avant le déploiement de la chaîne (indiqué par la présence d'une requête de mise à jour dans le module d'optimisation) alors l'orchestrateur procède à ladite mise à jour. Une fois que le réseau est prêt pour accueillir la chaîne calculée au terme du processus

d'orchestration, cette dernière est compilée et déployée sur les différents switches qui ont été identifiés pour lui assurer les meilleurs performances.

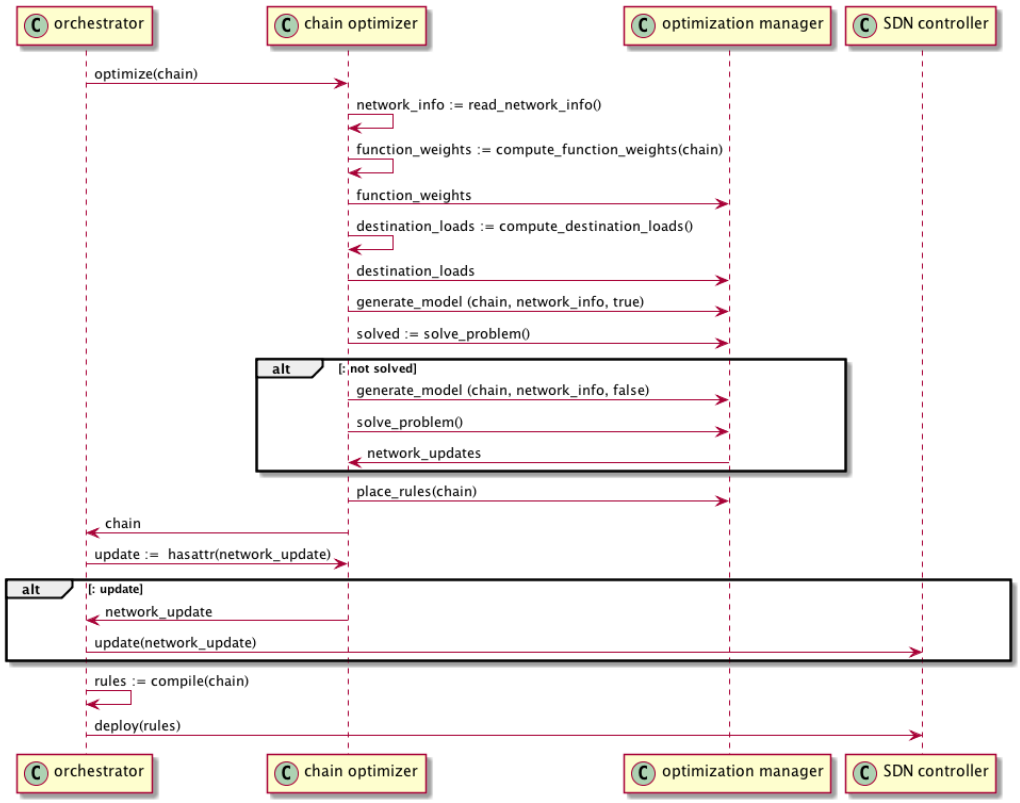

FIGURE 6.3 – Architecture pour l'optimisation de chaînes de fonctions de sécurité

6.2.2 Placement des règles d'une chaîne de fonctions de sécurité

Après la résolution de notre problème d'optimisation, nous disposons de la solution optimale pour le placement des règles d'une chaîne de fonctions de sécurité. Pour être exploitée cette solution doit être traduite de sorte à correspondre à la topologie du réseau dans lequel la chaîne de fonctions de sécurité devra être déployée. Pour ce faire il importe ici de concrétiser les abstractions qui avaient été précédemment utilisées pour minimiser la complexité du problème de placement. Plus précisément les lots de destinations doivent être substitués par les ensembles d'adresses IP les constituant, de même les chemins réseau doivent être remplacés par les listes d'identifiants de switches les constituant. Nous représentons ainsi la solution du problème de placement comme un dictionnaire indexé par les adresses IP dont chaque entrée contient la liste des identifiants de switches par lesquels le trafic généré pour une adresse doit passer.

A partir de ce dictionnaire, nous pouvons placer les règles de chaque fonction de sécurité dans une chaîne. Pour ce faire nous parcourons l'ensemble des fonctions constituant une chaîne, pour chacune d'entre elle nous collectons sa liste d'adresses IP, nous construisons l'ensemble de switches correspondant en fonction de la position de la fonction dans la chaîne. Enfin nous associons ses règles avec les switches correspondants.

Une fois opérée cette phase de placement des règles d'une chaîne de fonctions de sécurité reste encore à assurer la bonne redirection du trafic entre les différentes fonctions de sécurité. En effet, la non-linéarité du réseau nous oblige à considérer la possibilité qu'un switch soit suivi ou précédé de plusieurs autres équipements réseau. Les règles déployées sur chaque switch doivent ainsi être configurées de sorte à rediriger le trafic vers la bonne fonction de sécurité, ce chaînage doit être implémenté dans les deux sens, à savoir de l'équipement intelligent vers les serveurs distants et réciproquement.

Pour opérer la redirection des règles d'un switch vers un autre nous avons recours aux numéros de ports liant ces équipements. En effet dans la topologie décrivant le réseau que nous avons chargé au début de notre procédure d'optimisation se trouve la mention du numéro de port sur lequel chaque extrémité d'un canal doit être connecté.

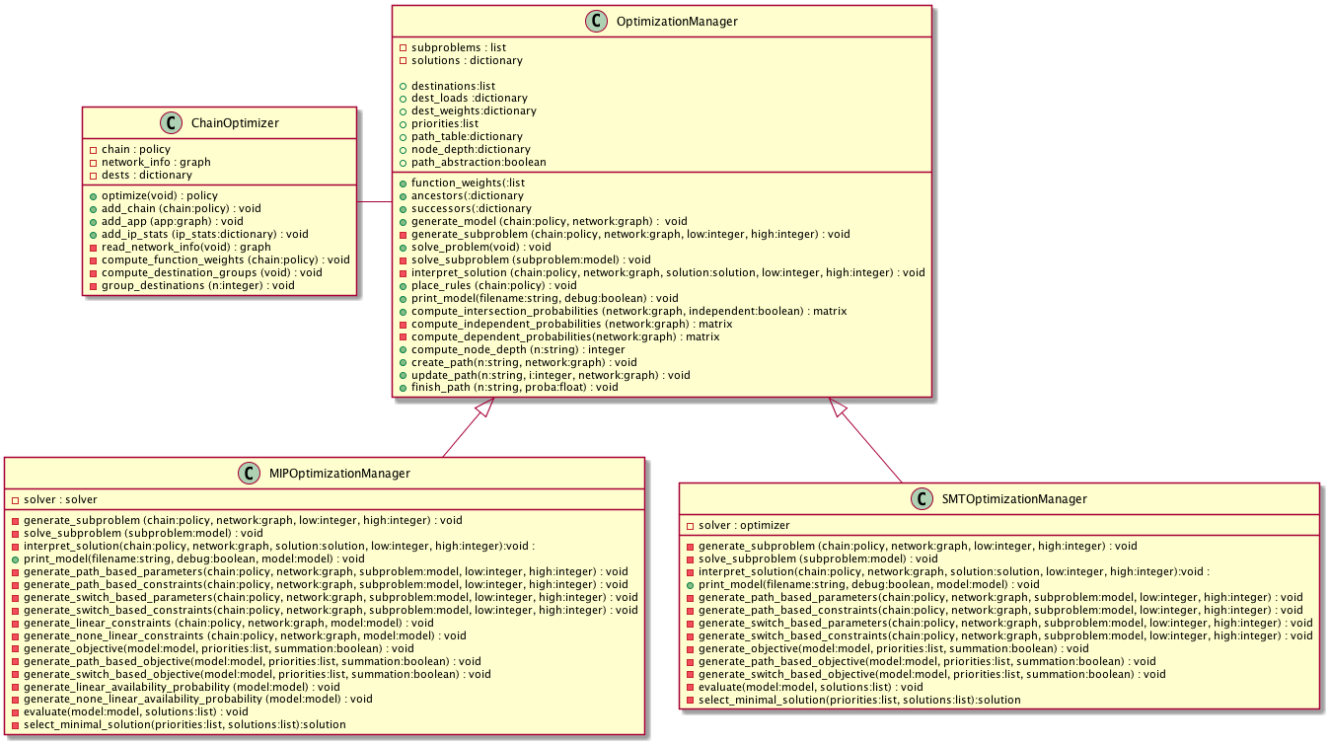

FIGURE 6.4 – Diagramme des classes de notre module d'optimisation de chaînes de sécurité

Cette information nous permet de repérer le numéro de port sur lequel doit être émis le trafic sortant de chaque switch en fonction du prochain switch à atteindre. Cette information est calculée en fonction de l'identifiant de switch associé à chaque règle de fonction de sécurité ainsi qu'en fonction des adresses IP concernées. Une fois identifiés les numéros de port devant être utilisés pour émettre le trafic sortant d'un switch ceux-ci sont utilisés pour configurer les règles de sécurité de la même manière qu'elles ont été précédemment placées.

6.3 Prototype et résultats expérimentaux

Nous avons évalué les performances de notre approche au travers d'expérimentations pratiques. Les calculs ont été réalisées sur un ordinateur portable Macbook Pro avec un processeur Intel Core i7 (2.5 GHz) et 16 GB de RAM. En termes de solvers d'optimisation nous avons utilisé le solver de simplex glpk encore connu comme glpsol [33] (v4.64), le solver MINLP couenne [65] (0.5.6), et νZ, l'extension de z3 (4.8.0) pour de l'optimisation basée sur SMT [21] qui ont déjà été abordés dans le chapitre décrivant les travaux relatifs de la thèse.

De sorte à comparer les performances de ces différentes méthodes d'optimisation, nous avons retenu de comparer l'influence du nombre de chemins dans le réseau ainsi que du nombre d'agrégats d'adresses IP à placer sur les différents solvers d'optimisation, nous avons mesuré leur temps de réponse ainsi que leur consommation mémoire. Nous avons considéré des exemples synthétiques plutôt que des cas d'étude de la vie réelle de sorte à mieux explorer les performances suivant les deux axes que nous avons défini, la variation du nombre d'adresses IP devant être placées induisant de facto une variation correspondante du nombre de règles.

6.3.1 Implémentation et prototype

Le travail présenté dans ce chapitre correspond au module ChainPlacement du graphique 3.8. Le système décrit dans ce chapitre a été implémenté en Python, les classes correspondantes sont décrites dans le diagramme 6.4. Il inclut un total de 2317 lignes de code, incluant l'implémentation de l'interface avec les différents solvers d'optimisation. Le calcul de la valeur des paramètres du modèle d'optimisation est assuré par la classe ChainOptimizer. Ces paramètres sont ensuite transmis à l'une des sous-classes d'OptimizationManager gérant respectivement l'optimisation basée sur simplex et MINLP pour MIPOptimizationManager et sur SMT pour SMTOptimizationManager.

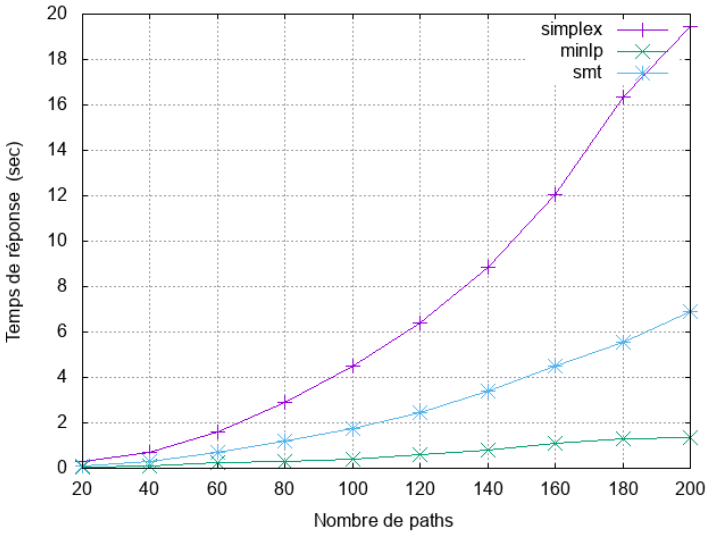

FIGURE 6.5 – Influence du nombre de chemins sur le temps de réponse

Cette architecture assure le placement des règles des différentes fonctions d'une chaîne sur les différents switches du réseau. Après cette opération les chaînes sont compilées sous forme de règles OpenFlow et déployées dans le réseau. Nous avons effectué des simulations du comportement des chaînes en utilisant l'environnement dédié MiniNet. Les règles générées pour nos chaînes sont correctement déployées dans le réseau et permettent d'émuler le trafic généré par les applications correspondantes.

6.3.2 Influence du nombre de chemins

Nous évaluons ici l'influence du nombre de chemins sur les performances des différents solvers d'optimisation. La figure 6.5 montre l'influence de ce paramètre sur le temps de réponse des différents solvers. Nous varions le nombre de chemins de 20 à 200 par incréments de 20. Concrètement, chaque chemin correspond à 5 switches SDN composés en séquence, donc notre expérimentation correspond à une variation de 100 à 1000 switches SDN avec un incrément de 100. Ces résultats semblent indiquer le MINLP solving comme le meilleur candidat pour résoudre le problème de placement. L'optimisation basée sur du SMT solving présente des performances raisonnables, alors que de manière surprenante les pires performances sont celles du simplex solver. Ceci peut être expliqué par la complexité du modèle impliquant la simulation des produits de variables binaires pour le calcul de la probabilité de disponibilité du réseau, conduisant à un nombre quadratique de contraintes à résoudre. En complément à ces expérimentations, nous avons encore mesuré l'influence du nombre de chemins sur la consommation mémoire des différents solvers, ces résultats sont présentés dans la figure 6.6.

Ces résultats indiquent que les bons temps de réponse du solver MINLP sont contrebalancés par une mauvaise consommation mémoire : ceci pourrait en rendre l'utilisation pratique en réseau SDN infaisable du fait que le contrôleur du réseau est déjà un goulot d'étranglement. La baisse observée dans cette courbe est probablement due à la mise en place de mécanismes de compression se déclenchant au-delà d'un certain seuil. En comparaison, la consommation mémoire du solver de simplex et de l'optimisation basée sur SMT sont très proches l'un de l'autre. Basé sur ces résultats, nous pouvons conclure que le SMT solving est un bon compromis pour résoudre de larges problèmes avec un bon temps de réponse et une consommation mémoire acceptable, Toutefois ces résultats ont encore besoin d'être confirmés par une évaluation pratique en réseau. Les solutions produites par les différentes méthodes de placement sont en outre bien comparables, c'est-à-dire que ces outils produisent les mêmes solutions pour les mêmes instances du problème.

6.3.3 Influence du nombre d'agrégats de destinations

Notre second critère d'évaluation est le nombre d'agrégats de destinations à surveiller. Pour ces expérimentations nous supposons un réseau comportant 20 chemins. Nous avons fait varier le nombre de lots de destinations de 5 à 50 avec un incrément de 5, les temps de réponse des solvers d'optimisation apparaissent dans la figure 6.7.

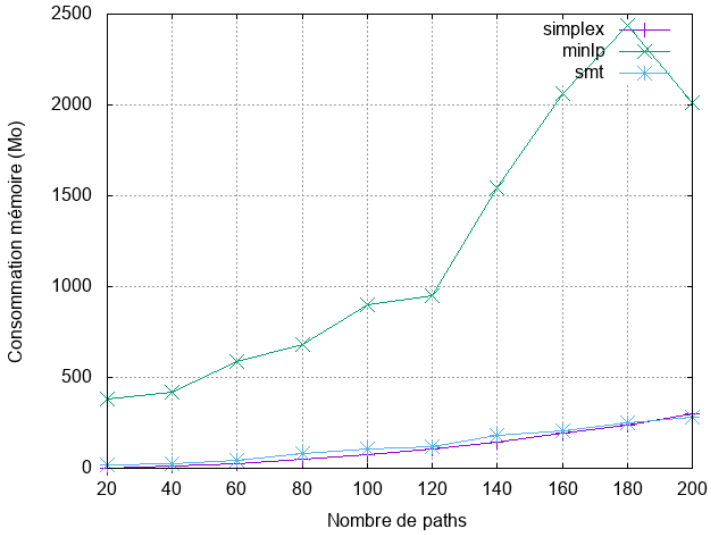

FIGURE 6.6 – Influence du nombre de chemins sur la consommation mémoire

Ces résultats jettent à nouveau un éclairage différent sur les performances des solvers d'optimisation. Le MINLP solving clairement présente les pires performances dans ce contexte, les performances de l'optimisation basée sur SMT sont un peu meilleures tandis que le simplex solving n'est pas affecté par le nombre d'agrégats à placer dans ces expérimentations. Ces observations semblent indiquer le simplex solver comme un bon candidat pour implémenter l'optimisation du placement lorsqu'il y a de nombreux lots de destinations à placer, au moins tant que le nombre de chemins reste relativement faible (cf. nos expériences précédentes). De sorte à confirmer ces résultats, nous avons évalué l'influence du nombre d'agrégats de destinations sur la consommation mémoire des différents solvers d'optimisation. Les résultats apparaissent dans la figure 6.8 et confirment nos conclusions précédentes. A nouveau nous observons que la consommation mémoire du MINLP solving est très coûteuse. L'optimisation basée sur SMT est relativement basse et le simplex solving clairement présente les meilleures performances pour ce critère.

Résumons, tous nos résultats illustrent le fait que la taille du modèle est un paramètre crucial pour minimiser et obtenir d'acceptables performances. Nous prévoyons de conduire un plus large ensemble d'expériences de sorte à vérifier que notre approche de placement passe à l'échelle dans des réseaux SDN réels.

6.4 Résumé

Dans ce chapitre, nous avons présenté nos travaux relatifs au placement d'une chaîne de fonctions de sécurité dans un réseau. Nous avons commencé par spécifier ledit problème en langage naturel avant d'en donner deux optimisations visant à en réduire la complexité. A partir de cette description informelle, nous avons spécifié les variables, paramètres et contraintes de notre modèle avant d'en définir les trois objectifs. Nous avons ensuite présenté l'algorithme d'association des règles d'une chaîne avec les adresses IP assignées par le module d'optimisation. Une fois placées, les règles d'une chaîne compilent effectivement vers des règles OpenFlow interprétables par le simulateur de réseau SDN MiniNet. Finalement, nous avons présenté l'évaluation des performances des trois outils de résolution que nous avons retenus de comparer : au terme de cette évaluation, il ressort que les différentes méthodes que nous avons considérées présentent différents avantages et inconvénients en fonction du contexte. Toutefois, il semble que les méthodes basées sur du SMT solving optimisant pourraient présenter un réel intérêt si elles faisaient l'objet de développements supplémentaires. En l'état actuel, ce sont néanmoins les méthodes de simplex solving qui s'imposent comme apportant les meilleures performances tant du point de vue du temps de réponse que de la consommation mémoire.

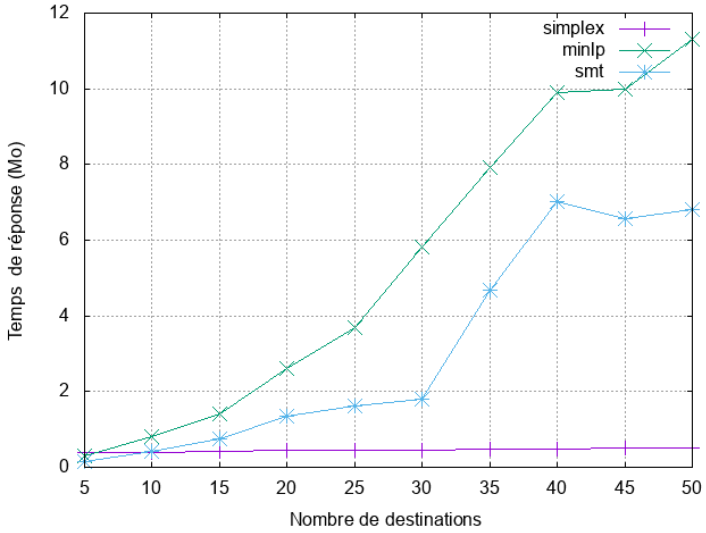

FIGURE 6.7 – Influence du nombre d'agrégats de destinations sur le temps de réponse

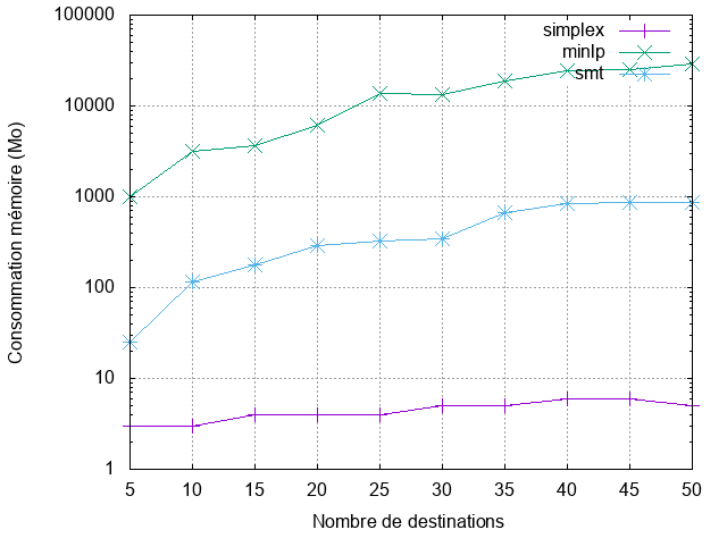

FIGURE 6.8 – Influence du nombre d'agrégats de destinations sur la consommation mémoire

7

Conclusions et perspectives

Nous rappelons au lecteur que notre travail porte sur la protection d'équipements intelligents tels que des smartphones ou des tablettes au travers du déploiement de chaînes de fonctions de sécurité. Il est organisé selon trois axes principaux : la spécification d'un orchestrateur responsable de la gestion des chaînes de fonctions de sécurité en environnement SDN, l'intégration de méthodes formelles pour la vérification ou la synthèse desdites chaînes, et enfin, l'optimisation du placement des chaînes en vue d'en minimiser l'impact sur les performances du réseau. Ce chapitre présente les conclusions générales de cette thèse. Nous commençons par dresser le bilan des contributions relatives à notre travail, avant de présenter comment ce dernier vient répondre à la problématique que nous avions introduite au chapitre 1. Finalement, nous présentons un ensemble de perspectives de recherche dans ce domaine.

7.1 Bilan des travaux de la thèse

Au travers de ce mémoire, nous avons synthétisé les résultats de nos travaux touchant à l'orchestration et à la vérification de fonctions de sécurité pour des environnements intelligents. En premier lieu, nous avons réalisé un état de l'art des travaux relatifs aux différentes méthodes abordées dans le cadre de cette thèse au sein du chapitre 2. Notre examen des solutions relatives à la sécurité des réseaux, et plus particulièrement des équipements intelligents, révèle un manque de vérifiabilité de la cohérence desdites politiques, et une difficulté pour établir le profil d'applications mobiles en temps réel. En complément, notre analyse des travaux relatifs à la programmabilité et virtualisation réseau a révélé une complexité accrue liée à une plus grande flexibilité, ainsi qu'à l'ignorance dans laquelle nous nous trouvons le plus souvent quant à l'implémentation des solutions supportant les fonctions de sécurité. Notre étude des différentes méthodes formelles a mis en avant la complexité algorithmique, ainsi que la difficulté corollaire pour les appliquer à la validation de politiques réseau complexes. Enfin, notre état de l'art a également traduit cette complexité algorithmique au regard des méthodes d'optimisation ou d'apprentissage de processus, qui peut rendre difficile leur exploitation pour le profilage d'applications mobiles en temps réel.

Nous avons présenté au chapitre 3 l'architecture de notre orchestrateur de chaînes de fonctions de sécurité. Celui-ci s'intègre dans un réseau de type SDN comme une application cliente du contrôleur, les chaînes sont ainsi déployées en utilisant les interfaces haut niveau que ce dernier met à disposition. Nous avons développé un prototype permettant l'expérimentation réaliste de nos différents travaux. Notre orchestrateur s'appuie sur un module d'apprentissage du comportement réseau d'applications Android, un module d'extraction de spécifications logiques de ces mêmes applications, un module de synthèse de chaînes utilisant un système d'inférence basé sur les clauses de Horn, un algorithme de factorisation de chaînes générées en parallèle, un module de vérification formelle validant la cohérence de la chaîne ainsi obtenue, et un module de placement des règles dans le réseau. Cette stratégie d'orchestration est conduite par l'apprentissage du modèles des applications à protéger. Nous nous sommes basés sur un modèle markovien des applications, ce modèle étant obtenu à partir des traces de communications réseau produites par une application. Nous avons discuté la pertinence de nos algorithmes d'agrégation de logs pour minimiser la complexité des algorithmes d'apprentissage et des modèles résultants. Finalement, nous avons introduit notre algorithme d'apprentissage et nous l'avons comparé avec les résultats produits par Synoptic et Invarimint. Au terme de ces travaux, il ressort que notre approche permet d'obtenir des modèles plus simples que ceux produits par Synoptic tout en étant plus expressifs que ceux produits par Invarimint.

Nous avons présenté ensuite notre approche de synthèse de chaînes utilisée par l'orchestrateur au chapitre 4. Celle-ci s'appuie sur un système d'inférences utilisant les clauses de Horn pour construire une chaîne de fonctions de sécurité à partir de la spécification logique des besoins de protection d'une application. Pour obtenir lesdits

besoins, nous avons introduit une méthode d'exploitation des automates markoviens présentés au chapitre 3. Nous avons complété ce modèle avec des propriétés dérivées des permissions requises par chaque application à protéger. Nous avons réalisé un prototype de cette approche, celui-ci a été intégré à l'implantation de notre orchestrateur. Nous avons démontré plusieurs propriétés assurées par notre système d'inférence, et avons mesuré expérimentalement que notre approche garantissait des performances de détection acceptables. Cette approche est complétée par une technique de factorisation des chaînes générées. Elle consiste à regrouper les règles de fonctions du même type, telles que par exemple celles relatives à des pare-feux ou des systèmes de détection d'intrusion. Nous avons introduit l'utilisation de cet algorithme, en comparaison avec deux approches naïves consistant respectivement à composer les chaînes en parallèle sans autre adaptation, ou à utiliser notre module de synthèse pour l'ensemble des besoins de sécurité de toutes les applications à protéger simultanément. Nous avons comparé les bénéfices introduits par notre technique quant à la simplicité des chaînes de sécurité ainsi regroupées. Il ressort de cette étude que notre approche permet de générer des chaînes plus simples que les deux approches naïves avec lesquelles nous l'avons comparée et qu'en outre notre architecture d'orchestration offre de bonnes performances qui en garantit la faisabilité en cas réel.

Nous avons détaillé nos travaux relatifs à la vérification formelle de chaînes de fonctions de sécurité pour un réseau SDN au chapitre 5. En particulier, nous avons proposé une approche fondée sur la vérification automatique du programme d'une chaîne de sécurité pour en assurer la validation avant le déploiement. Notre approche complète le travail présenté dans [63] en étendant la vérification du plan de contrôle supportée par Kinetic par une vérification du plan de données. Nous avons ainsi proposé des algorithmes de traduction du modèle d'une chaîne de sécurité en SMTlib ainsi que dans le langage d'entrée du model checker nuXmv. Nous avons enrichi ce travail d'un langage permettant la spécification de propriétés sur le trafic. En particulier, nous avons détaillé comment ce module peut être utilisé pour la recherche de contradictions dans les règles générées pour différentes chaînes synthétisées en parallèle. Ces contradictions peuvent ainsi être corrigées à partir des contre-exemples produits par les outils de résolution que nous avons considérés. Nous avons comparé les performances des différents outils de vérification formelle que nous avions intégrés dans notre prototype. Au sortir de ces expérimentations, il apparaît que le model checker nuXmv apporte de meilleures performances que les SMT solvers CVC4 ou veriT. Ces résultats pourraient toutefois être remis en cause par une modification de la modélisation utilisée pour représenter les chaînes. En effet, l'utilisation de mots binaires pour représenter les entités réseau reconnues par une chaîne pourrait potentiellement contribuer à diminuer la complexité des procédures de SMT solving.

Enfin, nous avons présenté nos travaux relatifs au placement de chaînes de fonctions de sécurité au sein d'un réseau SDN au chapitre 6. Nous avons introduit un modèle de placement faisant intervenir les adresses IP propres à chaque application, ainsi que la topologie sous-jacente du réseau. Pour simplifier ce modèle, nous en avons introduit deux abstractions, l'une visant à regrouper les adresses IP surveillées par une chaîne, et l'autre visant à minimiser le nombre d'emplacements potentiels pour une chaîne dans le réseau. Nous avons ensuite détaillé les paramètres, les variables, les contraintes et les objectifs. En particulier, nous avons présenté comment ce modèle devait être altéré afin de à répondre au problème réciproque de provisionnement de ressources, dans le cas où une chaîne ne pourrait pas être déployée dans le réseau. Nous avons terminé notre travail par le comparatif des performances apportées par différents solvers d'optimisation. Il ressort de ce comparatif que, si les travaux liés à la résolution de problèmes d'optimisation par application de méthodes formelles présentent un réel potentiel, il reste encore à en améliorer la complexité pour permettre leur pleine utilisation en vue d'une intégration en temps réel.

7.2 Liste des publications relatives à cette thèse

Nous reprenons ici les publications internationales résultant de notre travail de thèse, nous les citons par ordre chronologique :

- Nicolas Schnepf, Stephan Merz, Rémi Badonnel, and Abdelkader Lahmadi. Automated Verification of Security Chains in Software-Defined Networks with Synaptic. In *Proceedings of the 3rd IEEE Conference on Network Softwarization (NetSoft'17), Acceptance Rate of 18.6%*, 2017
- Nicolas Schnepf, Stephan Merz, Rémi Badonnel, and Abdelkader Lahmadi. Towards Generation of SDN Policies for Protecting Android Environments based on Automata Learning. In *Proceedings of the 16th Network Operations and Management Symposium (IEEE/IFIP NOMS'18), MC*, 2018
- Nicolas Schnepf, Stephan Merz, Rémi Badonnel, and Abdelkader Lahmadi. Synaptic : a Formal Checker for SDN-based Security Policies. In *Proceedings of the 16th Network Operations and Management Symposium (IEEE/IFIP NOMS'18), Demonstration*, 2018
- Nicolas Schnepf, Stephan Merz, Rémi Badonnel, and Abdelkader Lahmadi. Rule-Based Synthesis of Chains of Security Functions for Software-Defined Networks. In *Proceedings of the 18th International Workshop on Automated Verification of Critical Systems (AVOCS'18)*, 2018

— Nicolas Schnepf, Stephan Merz, Rémi Badonnel, and Abdelkader Lahmadi. Automated Factorization of Security Chains in Software-Defined Networks. In *Proceedings of the 16th IFIP/IEEE Symposium on Integrated Network and Service Management (IM 2019), MC*, 2019
— Nicolas Schnepf, Stephan Merz, Rémi Badonnel, and Abdelkader Lahmadi. A Tool Suite for the Automated Synthesis of Security Function Chains. In *Proceedings of the 16th IFIP/IEEE Symposium on Integrated Network and Service Management (IM 2019), Demonstration*, 2019

7.3 Réponses apportées à la problématique

Nous présentons ici les conclusions de notre étude dans le cadre de cette thèse. Pour rappel, les trois axes constitutifs de notre questionnement étaient la conception de l'architecture d'un orchestrateur responsable de la gestion des chaînes de fonctions de sécurité, l'identification des méthodes formelles les plus susceptibles de supporter la vérification de ces chaînes de sécurité en temps réel, et enfin la conception de stratégies d'optimisation permettant de minimiser le coût général du déploiement des chaînes dans le réseau.

7.3.1 Orchestration de fonctions réseau

Les éléments de réponse au premier axe de notre questionnement sont présentés dans le chapitre 3, où nous avons proposé l'architecture générale de notre orchestrateur de chaînes de fonctions de sécurité. En particulier, nous avons identifié des besoins en termes d'apprentissage du comportement d'applications, de détection de comportements malveillants à partir des modèles ainsi induits, de synthèse formelle basée sur des clauses de Horn pour permettre la génération des chaînes de fonctions de sécurité correspondantes, de factorisation pour assurer le regroupement desdites chaînes en une seule structure plus simple à gérer, de vérification formelle pour en assurer la cohérence et finalement d'optimisation pour en minimiser l'impact sur le réseau.

En particulier, nous avons conclu que dans le domaine des environnements intelligents, l'orchestration de fonctions de sécurité doit être conduite par l'identification des besoins spécifiques des utilisateurs. En effet, l'écosystème des applications relatives à ces environnements est très vaste et offre une grande diversité de profils, il est donc essentiel d'avoir recours à des méthodes d'apprentissage de processus pour permettre l'identification automatique des besoins d'un utilisateur. Cet apprentissage peut s'appuyer sur les communications réseau d'une application, il peut également exploiter les permissions régissant l'accès aux ressources d'un utilisateur.

Notre approche d'orchestration se décompose en deux phases, à savoir l'apprentissage du comportement d'applications qui est réalisé en amont, puis la synthèse des chaînes qui peut être réalisée en temps réel. Cette distinction repose sur la constatation que la phase de collecte de traces réseau et d'apprentissage de modèles correspondants prend beaucoup de temps, il est donc nécessaire d'y procéder de manière proactive afin de à permettre leur exploitation à l'exécution. Pour permettre ce découplage, nous nous sommes appuyés sur des solutions de stockage en base de données, ces dernières pourraient ainsi être utilisées pour enregistrer encore davantage d'informations relatives au comportement des applications.

7.3.2 Vérification formelle de chaînes de sécurité

Concernant le second axe de notre questionnement, nous avons apporté les éléments de réponse nécessaires à l'identification des méthodes formelles les plus adaptées à la validation des chaînes aux chapitres 4 et 5. Nous avons proposé d'avoir recours tant à des méthodes de synthèse que de vérification pour assurer la validation des chaînes. Utilisées conjointement, ces deux approches permettent une validation des propriétés devant être garanties pour la protection des utilisateurs, telle que par exemple la réponse à leurs besoins particuliers ou la compatibilité des règles assurant cette protection. Il convient désormais de procéder à des ajustements de la modélisation que nous utilisons avec chacune de ces méthodes de vérification, en vue d'en optimiser les performances en termes de temps de réponse.

Les méthodes de synthèse formelle apportent un réel intérêt dans l'exploitation des besoins de sécurité d'un utilisateur. En effet, l'extraction de ces besoins sous forme de prédicats logiques sur la base de méthodes de classification permet de générer à moindre coût une spécification logique pouvant servir de base axiomatique à une approche d'inférence. L'utilisation d'une telle approche permet ainsi la génération de chaînes spécifiquement adaptées aux besoins de chaque application, sans avoir à vérifier formellement la cohérence de chaînes pré-établies par l'opérateur d'un réseau.

Les méthodes de vérification formelle de chaînes de fonctions de sécurité peuvent toutefois introduire un gain dans la validation de multiples chaînes générées en parallèle pour la protection de plusieurs applications, par exemple en vérifiant leur compatibilité mutuelle. En effet, dans un tel contexte, il n'est pas dit que les chaînes ne contiennent pas de règles contradictoires du fait de l'isolation de leurs synthèses respectives. Dans ce cas, les

méthodes de vérification formelles peuvent être utilisées pour la détection et la résolution de conflits avant de procéder au déploiement des chaînes dans le réseau.

7.3.3 Optimisation du déploiement des chaînes de sécurité

Finalement, le troisième axe de notre questionnement portant sur la minimisation de l'impact du déploiement des chaînes de sécurité a trouvé ses réponses aux chapitres 4 et 6. Concernant la minimisation du nombre de règles et de fonctions de sécurité devant être déployées dans le réseau, nous avons montré que notre approche de factorisation permet de construire une chaîne de fonctions de sécurité minimale au point de vue de ces deux métriques. Une telle approche présente un intérêt tout particulier dans le contexte de la virtualisation de fonctions réseau où il est nécessaire de minimiser autant que possible le nombre d'instances de chaque fonction déployées dans le réseau.

Concernant l'optimisation du placement des chaînes, nous avons montré au chapitre 6 que notre stratégie de placement des règles sur les switches du réseau SDN permet de minimiser le coût du déploiement dans le réseau. Il peut toutefois être intéressant d'autoriser la réplication de certaines fonctions de sécurité en plusieurs points du réseau en vue d'améliorer la résilience des chaînes à d'éventuelles pannes. L'utilisation de méthodes formelles telles que les algorithmes de simplex, de MINLP solving ou de SMT solving optimisant présente également ici un intérêt non négligeable.

7.4 Perspectives de recherche

Notre travail dans le cadre de la présente thèse peut être complété et approfondi selon plusieurs directions que nous abordons ici en les mettant en lien avec les trois axes de notre travail de recherche.

7.4.1 Amélioration de la détection de comportements malveillants

Notre approche d'orchestration pourrait tout d'abord être affinée de plusieurs façons pour permettre une meilleure caractérisation des besoins des utilisateurs. En premier lieu, l'approche de détection de dénis de service, de scans de ports, de worms et de botnets que nous avons présentée au chapitre 4 pourrait être grandement améliorée par des travaux, en collaboration avec des chercheurs travaillant spécifiquement sur ces problématiques de caractérisation. L'architecture de notre orchestrateur permet l'intégration de nouvelles méthodes de détection sans avoir à en modifier profondément la structure générale, il serait donc tout à fait envisageable de considérer de nouvelles approches pour l'exploitation des flux réseau générés par une application.

Dans la même perspective, il semble intéressant de procéder à un approfondissement de l'exploitation que nous faisons des permissions requises par les applications Android. En particulier, des travaux touchant à la génération d'expressions régulières reconnaissant les données liées à chaque type de permissions nous semblent du plus grand intérêt. Ces expressions régulières pourraient ainsi être utilisées pour identifier de possibles fuites de données qui pourraient alors être prévenues par notre approche d'orchestration. En particulier, une telle approche nous permettrait d'affiner la spécification des règles de DLP que nous générons afin de à ce qu'elles prennent effectivement en compte le type des données transmises par une application. Dans le contexte de l'Internet contemporain où la majeure partie du trafic est chiffré, il serait nécessaire de mettre ce travail en lien avec des méthodologies d'analyse dédiées telles que celle présentée dans [96].

Enfin, il nous semble intéressant d'affiner la spécification des besoins relatifs à une application Android par la prise en compte de son comportement au niveau système. En effet, les flux de données lors de l'exécution d'une application peuvent révéler des comportements malveillants, il serait donc intéressant de disposer d'une approche de profilage prenant en compte ces aspects pour pouvoir les appréhender dans notre approche d'orchestration. Ces aspects pourraient être collectés par le même agent que celui que nous utilisons déjà pour la collecte de traces de flux, il faudrait ensuite mettre ces informations en lien par l'utilisation de timestamps ou de traces systèmes de l'émission de flux réseau. Un tel profilage pourrait par exemple être mis en lien avec le profil réseau que nous générons déjà ainsi qu'avec les permissions requises par une application et donc avec les données qu'elle manipule. Ces profils pourraient alors être stockés de la même manière que nous stockons les flux réseau générés par une application. Il serait alors intéressant d'envisager des approches de stockage permettant le rapprochement des comportements liés à plusieurs applications.

7.4.2 Exploitation et intégration de nouvelles méthodes formelles

L'utilisation de méthodes formelles dans notre approche d'orchestration pourrait être améliorée de plusieurs manières différentes. Premièrement, il pourrait être intéressant d'explorer l'utilisation de logiques plus expressives

que la logique du premier ordre pour la représentation des attaques. En particulier, les logiques temporelles nous semblent porteuses d'un potentiel prometteur pour capturer les aspects liés à la dynamique de tels comportements réseau. Il serait ainsi possible de procéder dans un premier temps à des travaux théoriques touchant à la modélisation des attaques traitées dans cette thèse au moyen de logiques temporelles telles que CTL ou LTL. Après cela, il serait pertinent de mettre ces modèles en lien avec les travaux relatifs à la détection d'attaques que nous évoquions plus haut pour permettre la génération automatique de traces du comportement d'applications malveillantes. En particulier, nous pourrions explorer les apports de méthodes de *runtime verification*, en particulier les approches de monitoring de propriétés temporelles au cours d'une exécution.

Dans la continuité de ce travail sur les logiques temporelles, il serait nécessaire de procéder à une extension de notre système d'inférence pour prendre en compte lesdites logiques. Les travaux relatifs à l'inférence en logiques d'ordre supérieur sur la base de clauses de Horn dédiées pourraient ici être du plus grand intérêt [74]. Ces travaux pourraient notamment permettre l'amélioration de la spécification des chaînes de fonctions de sécurité avec des aspects liés à leur dynamique. Une telle extension permettrait d'appréhender la génération de plan de données à états, par opposition à notre approche qui s'est focalisée sur le cas sans état. Ces travaux pourraient notamment comprendre la génération de configurations dédiées du plans de contrôle, par exemple en spécifiant la mise à jour attendue en cas de la découverte d'une attaque.

Concernant nos travaux relatifs à la vérification formelle de chaînes de sécurité présentés au chapitre 5, il pourrait être intéressant d'étendre le langage de propriétés, par exemple pour prendre en compte les aspects liés à la dynamique des chaînes. Il pourrait enfin être envisagé d'étendre notre modèle afin de à y prendre en compte pleinement les modèles des applications à protéger au travers de méthodes de model checking probabiliste dont il serait toutefois nécessaire de prendre la très haute complexité en compte. Une telle extension pourrait permettre d'intégrer des aspects quantitatifs dans notre procédure de vérification, le model checking probabiliste pouvant en effet être utilisé pour effectuer l'évaluation des performances d'un système.

7.4.3 Performances des chaînes de fonctions de sécurité

Il est envisageable d'étendre la portée de nos travaux sur l'optimisation du déploiement des chaînes, en prenant en compte plusieurs aspects visant à améliorer leurs performances dans le réseau. Concernant l'implémentation de nos chaînes, nous avons exploré plusieurs langages dédiés à la spécification de politiques SDN, il pourrait être intéressant d'étendre ce comparatif en vue d'identifier des implémentations assurant de meilleures performances. Nous avons démontré l'agnosticisme de notre approche quant au langage utilisé pour implanter les chaînes, l'intégration de nouvelles représentations ne devrait donc pas demander d'efforts excessifs. Il serait en outre possible d'exploiter les possibilités mises à disposition par des protocoles tels qu'OpenFlow pour appréhender de nouvelles métriques.

Il reste également à comparer les performances apportées par une implantation basée uniquement sur les switches SDN, avec une implantation utilisant les technologies de virtualisation de fonctions réseau, dont il a été question dans l'état de l'art. Une telle extension pourrait par exemple permettre d'introduire davantage de programmabilité dans le réseau, en utilisant des fonctions supportant un langage de configuration dédié. Il serait alors envisageable d'utiliser nos travaux sur la génération de règles pour configurer automatiquement ces fonctions en vue d'assurer la détection d'attaques ou de fuites de données au niveau du réseau sans avoir recours au contrôleur.

Pour terminer, il nous semble également opportun d'étendre notre module d'optimisation pour en permettre une exploitation plus poussée. Il serait par exemple intéressant d'appréhender la reconfiguration du déploiement d'une chaîne de fonctions de sécurité après la panne d'un switch ou d'un lien dans le réseau, afin de à améliorer la résilience de notre approche d'orchestration. Il serait aussi intéressant d'envisager l'influence d'autres paramètres, notamment pour comparer le déploiement des chaînes sur des fonctions réseau virtualisées par rapport à des switches SDN. Une telle extension permettrait la génération d'un plan de déploiement qui combine les deux approches pour une meilleure optimisation des ressources du réseau.

A

Spécification SMTlib d'une politique de pare-feux distribués

```
(set-option :print-success false)
(set-option :produce-models true)
(set-logic QF_LIA)

; variables

(declare-fun allowed() Bool)
(declare-fun srcip() Int)
(declare-fun dstport() Int)
(declare-fun srcport() Int)

; values

(declare-fun ANY() Int)
(declare-fun ip1() Int)
(declare-fun ip0() Int)
(declare-fun port6() Int)
(declare-fun port7() Int)
(declare-fun port8() Int)
(declare-fun port3() Int)
(declare-fun port5() Int)
(declare-fun port0() Int)
(declare-fun port1() Int)
(declare-fun port4() Int)
(declare-fun port2() Int)

(assert (and (distinct ANY port3 port5 port0 port7 ip1 port8 port1 port4 port6
    port2 ip0) (= allowed (and (or (and (or (= srcip ip0) (= srcip ip1)) (or (=
    srcport port0) (= srcport port1) (= srcport port2))) (and (not (or (= srcip
    ip0) (= srcip ip1))) (or (= srcport port3) (= srcport port4) (= srcport
    port5)))) (or (= dstport port6) (= dstport port7) (= dstport port8)))) (and
    (or (and (or (= srcport port0) (= port1 srcport) (= port2 srcport)) (or (=
    srcip ip0) (= srcip ip1)) (or (= dstport port7) (= port6 dstport) (= dstport
     port8))) (and (not (or (= srcip ip0) (= srcip ip1))) (or (= srcport port3)
    (= port4 srcport) (= srcport port5)) (or (= dstport port7) (= port6 dstport)
     (= dstport port8)))) (not allowed))))
(check-sat)
(get-model)
```

```
(exit)
```

Listing A.1 – Modèle SMTlib complet pour l'exemple jouet du chapitre 5

B

Spécification nuXmv d'une politique de pare-feux distribués

```
MODULE main
VAR
state : {S0,S1,S2,S3,S4,S5,S6,S7};
FROZENVAR
srcip : {ANY,ip1,ip0};
dstport : {ANY,port6,port7,port8};
srcport : {ANY,port3,port5,port0,port1,port4,port2};
ASSIGN
init(state) := S0;
init(srcip) := {ip1,ip0};
init(dstport) := {port7,port6,port8};
init(srcport) := {port3,port5,port0,port1,port4,port2};
next(state) := case
(state=S0) & srcip in {ip1,ip0} : S1;
(state=S1) & srcip in {ip1,ip0} : S2;
(state=S2) & srcport in {port1,port0,port2} : S3;
(state=S0) & srcip in {ip1,ip0} : S4;
(state=S0) & ! (srcip in {ip1,ip0}) : S4;
(state=S4) & srcport in {port3,port4,port5} : S5;
(state=S0) & srcport in {port3,port5,port4,port1,port0,port2} : S1;
(state=S3|state=S5) & srcport in {port3,port5,port4,port1,port0,port2} : S1;
(state=S1) & dstport in {port6,port7,port8} : S7;
(state=S1) & dstport in {port6,port7,port8} : S7;
TRUE : state;
esac;
SPEC AG((((srcip=ip0|srcip=ip1) & (srcport=port1|srcport=port0|
srcport=port2) & (dstport=port7|dstport=port6|dstport=port8))) -> EF state=S7)
SPEC AG((((!((srcip=ip0|srcip=ip1)) & (srcport=port4|srcport=port3|
srcport=port5) & (dstport=port7|dstport=port6|dstport=port8))) -> EF state=S7)
-- This module uses the following abstractions :
--
-- port3=4000
-- port5=6000
-- port0=1000
-- port7=8000
-- ip1=253.182.3.14
-- port8=9000
-- port1=2000
```

```
-- port4=5000
-- port6=7000
-- port2=3000
-- ip0=198.122.37.15
--
```

Listing B.2 – Modèle nuXmv complet pour l'exemple jouet du chapitre 5

Bibliographie

[1] Android permissions's system. `https://developer.android.com/guide/topics/security/permissions.html`.

[2] Ehab Al-Shaer and Saeed Al-Haj. FlowChecker, Configuration Analysis and Verification of Federated OpenFlow Infrastructures. In *Proceedings of the 3rd ACM Workshop on Assurable and Usable Security Configuration (CCS'10)*, 2010.

[3] Ehab S. Al-Shaer and Hazem H. Hamed. Discovery of Policy Anomalies in Distributed Firewalls. In *Proceedings of the Twenty-third Annual Joint Conference of the IEEE Computer and Communications (INFOCOM'04)*, 2004.

[4] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *FMCAD*, pages 1–8. IEEE, 2013.

[5] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2) :87–106, November 1987.

[6] Joao Antunes, Nuno Neves, and Paulo Verissimo. Reverse engineering of protocols from network traces. In *Proceedings of the 2011 18th Working Conference on Reverse Engineering*, WCRE '11, pages 169–178, Washington, DC, USA, 2011. IEEE Computer Society.

[7] Irina Mariuca Asavoae, Jorge Blasco, Thomas M. Chen, Harsha Kumara Kalutarage, Igor Muttik, Hoang Nga Nguyen, Markus Roggenbach, and Siraj Ahmed Shaikh. Towards automated android app collusion detection. In *Proc. 1st Intl. Wsh. Innovations in Mobile Privacy and Security (IMPS 2016)*, volume 1575 of *CEUR Workshop Proceedings*, pages 29–37, London, UK, 2016.

[8] unknown author. smtlib, the satisfiability modulo theory library. `http://smtlib.cs.uiowa.edu`.

[9] Zafar Ayyub and Rui Miao. Simple-fying middlebox policy enforcement using sdn. In *ACM SIGCOMM Computer Communication Review*, 2013.

[10] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Octeau, and Sebastian Weisgerber. On demystifying the android application framework : re-visiting android permission specification analysis. In *Proceedings of the 25th USENIX Security Symposium (NSDI 2016)*, 2016.

[11] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. Vericon : Towards Verifying Controller Programs in Software-Defined Networks. In *Proc. 35th ACM SIGPLAN Intl. Conf. Programming Language Design (PLDI'14)*, pages 282–293, Edinburgh, UK, 2014.

[12] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for bit-vector arithmetic. In *Proceedings of the 35th Annual Design Automation Conference*, DAC '98, pages 522–527, New York, NY, USA, 1998. ACM.

[13] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009.

[14] Dominique Barthel, JP Vasseur, Kris Pister, Mijeom Kim, and Nicolas Dejean. Routing Metrics Used for Path Calculation in Low-Power and Lossy Networks. RFC 6551, March 2012.

[15] Ryan Beckett. *Network Control Plane Synthesis and Verification.* PhD thesis, University of Princeton, 2018.

[16] Ivan Beschastnikh, Jenny Abrahamson, Yuriy Brun, and Michael D. Ernst. Synoptic : Studying Logged Behavior with Inferred Models. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 448–451, New York, NY, USA, 2011. ACM.

[17] Ivan Beschastnikh, Yuriy Brun, Jenny Abrahamson, Michael D. Ernst, and Arvind Krishnamurthy. Using Declarative Specification to Improve the Understanding, Extensibility, and Comparison of Model-Inference Algorithms. In *IEEE Transactions on Software Engineering*, volume 41, pages 408–428, 2015.

[18] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, and Arvind Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with csight. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 468–479, New York, NY, USA, 2014. ACM.

[19] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsch. *Handbook of satisfiability.* IO press, 2008.

[20] A.W. Biermann and J.A. Feldman. On the Synthesis of Finite-State Machines from Samples of Their Behavior. In *IEEE Transactions on Computers*, 1972.

[21] Nikolaj Bjorner, Anh-Dung Phan, and Lars Fleckenstein. vz - an optimizing smt solver. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9035*, pages 194–199, Berlin, Heidelberg, 2015. Springer-Verlag.

[22] Jeremiah Blendin, Julius Rückert, Nicolai Leymann, Georg Schyguda, and David Hausheer. Position paper : software defined network services chaining. In *Proceedings of the Third European Workshop on Software Defined Networks (EWSDN 2014)*, 2014.

[23] Scott W. Brim and Brian E. Carpenter. Middleboxes : Taxonomy and Issues. RFC 3234, February 2002.

[24] Marco Canini, Daniele Venzano, Peter Pereskini, Dejan Kostic, and Rexford Jennifer. A Nice Way to Test OpenFlow Applications. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI'12)*, 2012.

[25] Rafael C. Carrasco and José Oncina. Learning stochastic regular grammars by means of a state merging method. In Rafael C. Carrasco and José Oncina, editors, *2nd Intl. Coll. Grammatical Inference and Applications (ICGI-94)*, volume 862 of *LNCS*, Alicante, Spain, 1994. Springer.

[26] N.M. Mosharaf Kabir Chowdhury and Raouf Boutaba. A survey of network virtualization. *Comput. Netw.*, 54(5) :862–876, April 2010.

[27] Benoit Claise. Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information. RFC 5101, January 2008.

[28] Benoît Claise. Cisco Systems NetFlow Services Export Version 9. RFC 3954, October 2004.

[29] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2) :244–263, April 1986.

[30] Edmund M. Clarke. The birth of model checking. In *25 Years of Model Checking*, volume 5000 of *LNCS*, pages 1–26. Springer, 2008.

[31] Jayanthkumar Cui, Weidong Kannan and Helen Wang. Discoverer : Automatic protocol reverse engineering from network traces. In *Proceedings of the 16th USENIX Security Symposium (NSDI 2017*, 2017.

[32] Shuaifu Dai, Alok Tongaonkar, Xiaoyin Wang, Antonio Nucci, and Dawn Song. Networkprofiler : Towards automatic fingerprinting of android apps. In *Proceedings of IEEE InfoCom 2013 (INFOCOM 2013)*, 2013.

[33] George B. Dantzig. *Linear programming and extensions.* Rand Corporation Research Study. Princeton Univ. Press, Princeton, NJ, 1963.

[34] George B. Dantzig. A history of scientific computing. chapter Origins of the Simplex Method, pages 141–151. ACM, New York, NY, USA, 1990.

[35] Sreerupa Das and Michael C Mozer. A unified gradient-descent/clustering architecture for finite state machine induction. In J. D. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems 6*, pages 19–26. Morgan-Kaufmann, 1994.

[36] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7) :394–397, July 1962.

[37] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3) :201–215, July 1960.

[38] Joseph Dilip and Stoica Ion. Modeling middle boxes. In *IEEE Network : The Magazine of Global Internetworking archive*, 2008.

[39] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan. Android Security, a Survey of Issues,
Malware Penetrations and Defenses. In *IEEE Communications Surveys & Tutorials*, 2015.

[40] Seyed K. Fayaz and Vyas Sekar. Testing Stateful and Dynamic Data Plane with FlowTest. In *Proceedings of the third Workshop on Hot Topics in Software-Defined Networking (HotSDN'14)*, 2014.

[41] Nick Feamster and Hyojoon Kim. Software-Defined Networks : Improving Network Management with SDN. In *IEEE Communications Magazine*, 2013.

[42] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The Road to SDN, an Intellectual History of Programmable Networks. *SIGCOMM Computer Communication Review*, 44(2) :87–98, 2014.

[43] V. Forejt, M. Kwiatkowska, G. Norman, and D. Parker. Automated verification techniques for probabilistic systems. In M. Bernardo and V. Issarny, editors, *Formal Methods for Eternal Networked Software Systems (SFM'11)*, volume 6659 of *LNCS*, pages 53–113, Bertinoro, Italy, 2011. Springer.

[44] Nate Foster, Michael J. Freedman, Arjun Guha, Rob Harrison, Naga Praveen Kata, Christopher Monsanto, Joshua Reich, Mark Reitblatt, Rexford Jennifer, Cole Schlesinger, Alec Story, and David Walker. Languages for Software-Defined Networks. In *Software Technology Group*, 2016.

[45] Nate Foster, Michael J. Freedman, Rob Harrison, Christopher Monsanto, and David Walker. Frenetic, a Network Programming Language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP'11)*, 2011.

[46] Nate Foster, Jedidiah McClurg, Hossein Hojjat, and Pavol Cerny. Efficient Synthesis of Network Updates. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, 2015.

[47] Nate Foster, Christopher Monsanto, Joshua Reich, Rexford Jennifer, and David Walker. Composing software defined networks. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation (NSDI 2013)*, 2013.

[48] Jérôme François, Lautaro Dolberg, Olivier Festor, and Thomas Engel. Network Security through Software Defined Networking : a Survey. In *IIT Real-Time Communications (RTC) Conference - Principles, Systems and Applications of IP Telecommunications (IPTComm)*, Chicago, United States, September 2014. ACM.

[49] S. García, M. Grill, J. Stiborek, and A. Zunino. An empirical comparison of botnet detection methods. *Comput. Secur.*, 45 :100–123, September 2014.

[50] GData. Mobile malware report 2018. `http://www.gdatasoftware.com`.

[51] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. Checking app behavior against app description. In *Proceedings of international Conference of Software Engineering (ICSE 2014)*, 2014.

[52] Mark J. Handley, Eric Rescorla, and Internet Architecture Board. Internet Denial-of-Service Considerations. RFC 4732, December 2006.

[53] Frank L. Hitchcock. The distribution of a product from several sources to numerous localities. *Journal of Mathematics and Physics*, 20(1-4) :224–230, 1941.

[54] Alfred Horn. On sentences which are true of direct unions of algebras. *J. Symb. Log.*, 16(1) :14–21, 1951.

[55] Hongxin Hu, Wonkyu Han, Gail Joon Ahn, and Ziming Zhao. Flowgard : building robust firewalls for software defined networks. In *Proceedings of the third workshop on Hot topics in software defined networking (SIGCOMM 2014)*, 2014.

[56] Gaëtan Hurel, Rémi Badonnel, Abdelkader Lahmadi, and Olivier Festor. Behavioral and Dynamic Security Functions Chaining for Android Devices. In *Proceedings of the 11th IFIP/IEEE/ACM SIGCOMM International Conference on Network and Service Management (CNSM'15)*, 2015.

[57] Gaëtan Hurel, Rémi Badonnel, Abdelkader Lahmadi, and Olivier Festor. Towards Cloud Based Compositions of Security Functions for Mobile Devices. In *IFIP/IEEE International Symposium on Integrated Network Management (IM'15)*, 2015.

[58] IDC. Worldwide quarterly mobile phone tracker 2018. `http://www.idc.com/tracker`.

[59] Mi-Young Kang, Jin-Young Choi, Inhye Kang, Hee Hwan Kwak, So Jin Ahn, and Myung-Ki Shin. *A Verification Method of SDN Firewall Applications.* IEICE Transactions on Communications, 2016.

[60] L. V. Kantorovich. Mathematical methods of organizing and planning production. *Manage. Sci.*, 6(4) :366–422, July 1960.

[61] Ahmed Khurshid, Xuan Zou, Winxuan Zhou, Matthew Caesar, and P. Brighten. VeriFlow : Verifying Network-wide Invariants in Real Time. In *Proceedings of the first Workshop on Hot Topics in Software-Defined Networks (HotSDN'12)*, 2012.

[62] Caner Kilinc, Todd Booth, and Karl Andersson. Walldroid : cloud assisted virtualized application specific firewalls for the android os. In *Proceedings of IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications (TRUSTCOM 2012)*, 2012.

[63] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhamad Shahbaz, Nick Feamster, and Russ Clark. Kinetic : Verifiable Dynamic Network Control. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15)*, 2015.

[64] Jeongmin Kim, Hyunwoo Choi, Hun Namkung, Woohyun Choi, Byungkwon Choi, Hyunwook Hong, Yongdae Kim, Jonghyup Lee, and Dongsu Han. Enabling automatic protocol behavior analysis for android applications. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, pages 281–295, New York, NY, USA, 2016. ACM.

[65] Jan Kronqvist, David E. Bernal, Andreas Lundell, and Ignacio E. Grossmann. A review and comparison of solvers for convex minlp. In *Springer US*, 2018.

[66] H. Kuai, F. Alajaji, and G. Takahara. A lower bound on the probability of a finite union of events. *Discrete Math.*, 215(1-3) :147–158, March 2000.

[67] Mariantonietta La Polla, Fabio Martinelli, and Daniele Sgandurra. A Survey on Security for Mobile Devices. In *IEEE Communications Surveys & Tutorials*, 2012.

[68] Abdelkader Lahmadi, Frederic Beck, Eric Finickel, and Olivier Festor. A platform for the analysis and visualization of network flow data of android environments. IFIP/IEEE International Symposium on Integrated Network Management (IM), May 2015. Poster.

[69] Tianyi Liang, Andrew Reynolds, Nestan Tsiskaridze, Cesare Tinelli, Clark Barrett, and Morgan Deters. An efficient smt solver for string constraints. *Form. Methods Syst. Des.*, 48(3) :206–234, June 2016.

[70] Adetokunbo A.O. Makanju, A. Nur Zincir-Heywood, and Evangelos E. Milios. Clustering event logs using iterative partitioning. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 1255–1264, New York, NY, USA, 2009. ACM.

[71] Gary S. Malkin and Tracy LaQuey Parker. Internet Users' Glossary. RFC 1392, January 1993.

[72] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon J. Ross, and Gianluca Stringhini. Mamadroid : Detecting android malware by building markov chains of behavioral models. *CoRR*, abs/1612.04433, 2016.

[73] Nick McKeown and Guru Parulkar. openflow, enabling innovation in campus networks. In *ACM SIGCOMM Computer Communication Review*, 2008.

[74] Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, New York, NY, USA, 1st edition, 2012.

[75] Sumit Nain and Moshe Y. Vardi. Branching vs. linear time : Semantical perspective. In *Proceedings of the 5th International Conference on Automated Technology for Verification and Analysis*, ATVA'07, pages 19–34, Berlin, Heidelberg, 2007. Springer-Verlag.

[76] Ricardo Neisse, Gary Steri, and Gianmarco Baldini. Enforcement of security policy rules for the internet of things. In *Proceedings of the 10th IEEE International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob'14)*, 2014.

[77] Jon Oberheide, Kaushik Veeraraghavan, Evan Cooke, Jason Flinn, and Farnam Jahanian. Virtualized in-cloud security services for mobile devices. In *Proceedings of the First Workshop on Virtualization in Mobile Computing*, MobiVirt '08, pages 31–35, New York, NY, USA, 2008. ACM.

[78] Jon Oberheide, Kaushik Veeraraghavan, Evan Cooke, Jason Flinn, and Farnam Jahanian. Virtualized in-cloud security services for mobile devices. In *Proceedings of the first workshop on virtualization in mobile computing (MOBIVIRT 2008)*, 2008.

[79] Andrés F. Ocampo, Juliver Gil-Herrera, Pedro H. Isolani, Miguel C. Neves, Juan F. Botero, Steven Latré, Lisandro Zambenedetti, Marinho P. Barcellos, and Luciano P. Gaspary. Optimal Service Function Chain Composition in Network Functions Virtualization. In *Proceedings of the IFIP International Conference on Autonomous Infrastructure, Management and Security (IFIP AIMS'17)*, pages 62–76. Springer International Publishing, 2017.

[80] Tony Ohmann, Michael Herzberg, Sebastian Fiss, Armand Halbert, Marc Palyart, Ivan Beschastnikh, and Yuriy Brun. Behavioral resource-aware model inference. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 19–30, New York, NY, USA, 2014. ACM.

[81] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permission demystified. In *dings of the 18th ACM conference on Computer and communications security (CCS 2011)*, 2011.

[82] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akllia, Sujata Banergee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. Pga : using graphs to express and automatically

reconcile network policies. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data communication (SIGCOMM 2015)*, 2015.

[83] Jingjing Ren, Ashwin Rao, Martina Lindorfer, Arnaud Legout, and David R. Choffnes. Recon : Revealing and controlling privacy leaks in mobile network traffic. *CoRR*, abs/1507.00255, 2015.

[84] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1) :23–41, January 1965.

[85] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*. Elsevier Science Inc., New York, NY, USA, 2006.

[86] A. Rozinat, M. Veloso, and W.M.P. van der Aalst. Evaluating the Quality of Discovered Process Models. In W. Bridewell, T. Calders, A.K. de Medeiros, S. Kramer, M. Pechenizkiy, and L. Todorovski, editors, *Proceedings of the ECML-PKDD Workshop on Induction of Process Models (IPM08)*, pages 45–52. University of Antwerp, Belgium, 2008.

[87] A. Sapio, M. Liao, Y. andBaldi., G. Ranjan, F. Risso, and A. Tongaonkar. Per-user policy enforcement on mobile apps through network virtualization. In *Proceedings of the 9th ACM workshop on Mobility in the evolving internet (MOBIARCH 2014)*, 2014.

[88] Nicolas Schnepf, Stephan Merz, Rémi Badonnel, and Abdelkader Lahmadi. Automated Verification of Security Chains in Software-Defined Networks with Synaptic. In *Proceedings of the 3rd IEEE Conference on Network Softwarization (NetSoft'17), Acceptance Rate of 18.6%*, 2017.

[89] Nicolas Schnepf, Stephan Merz, Rémi Badonnel, and Abdelkader Lahmadi. Rule-Based Synthesis of Chains of Security Functions for Software-Defined Networks. In *Proceedings of the 18th International Workshop on Automated Verification of Critical Systems (AVOCS'18)*, 2018.

[90] Nicolas Schnepf, Stephan Merz, Rémi Badonnel, and Abdelkader Lahmadi. Synaptic : a Formal Checker for SDN-based Security Policies. In *Proceedings of the 16th Network Operations and Management Symposium (IEEE/IFIP NOMS'18), Demonstration*, 2018.

[91] Nicolas Schnepf, Stephan Merz, Rémi Badonnel, and Abdelkader Lahmadi. Towards Generation of SDN Policies for Protecting Android Environments based on Automata Learning. In *Proceedings of the 16th Network Operations and Management Symposium (IEEE/IFIP NOMS'18), MC*, 2018.

[92] Nicolas Schnepf, Stephan Merz, Rémi Badonnel, and Abdelkader Lahmadi. A Tool Suite for the Automated Synthesis of Security Function Chains. In *Proceedings of the 16th IFIP/IEEE Symposium on Integrated Network and Service Management (IM 2019), Demonstration*, 2019.

[93] Nicolas Schnepf, Stephan Merz, Rémi Badonnel, and Abdelkader Lahmadi. Automated Factorization of Security Chains in Software-Defined Networks. In *Proceedings of the 16th IFIP/IEEE Symposium on Integrated Network and Service Management (IM 2019), MC*, 2019.

[94] Roberto Sebastiani and Patrick Trentin. Optimathsat : A tool for optimization modulo theories. *Journal of Automated Reasoning*, Dec 2018.

[95] Justine Sherry and Arvind Krishnamurthy. Making middleboxes someone else's problem : network processing as a cloud service. In *ACM SIGCOMM Computer Communication Review*, 2012.

[96] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Blindbox : Deep packet inspection over encrypted traffic. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 213–226, New York, NY, USA, 2015. ACM.

[97] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3) :733–749, July 1985.

[98] Anna Sperotto. *Flow-based Intrusion Detection*. PhD thesis, University of Twente, 2010.

[99] Xuetao Wei. Profiledroid : Multi-layer profiling of android applications. In *Proceedings of the 18th annual international conference on Mobile computing and networking (MOBICOM 2012)*, 2012.

[100] Mingyuan Xia, Lu Gong, Yuanhao Lyu, Zhengwei Qi, and Xue Liu. Effective real-time android application auditing. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 899–914, Washington, DC, USA, 2015. IEEE Computer Society.

[101] Mu Zhang, Yue Duan, Qian Feng, and Heng Yin. Towards automatic generation of security centric descriptions for android apps. In *dings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS 2015)*, 2015.

[102] Yunhui Zheng, Vijay Ganesh, Sanu Subramanian, Omer Tripp, Murphy Berzish, Julian Dolby, and Xiangyu Zhang. Z3str2 : An efficient solver for strings, regular expressions, and length constraints. *Form. Methods Syst. Des.*, 50(2-3) :249–288, June 2017.

[103] Saman Zonouz, Amir Houmansadr, Robin Berthier, Nikita Borisov, and William Sanders. Secloud : A cloud-based comprehensive and lightweight security solution for smartphones. *Comput. Secur.*, 37 :215–227, September 2013.

Résumé

Les équipements intelligents, notamment les smartphones, sont la cible de nombreuses attaques de sécurité. Par ailleurs, la mise en oeuvre de mécanismes de protection usuels est souvent inadaptée du fait de leurs ressources fortement contraintes. Dans ce contexte, nous proposons d'utiliser des chaînes de fonctions de sécurité qui sont composées de plusieurs services de sécurité, tels que des pare-feux ou des antivirus, automatiquement configurés et déployés dans le réseau. Cependant, ces chaînes sont connues pour être difficiles à valider. Cette difficulté est causée par la complexité de ces compositions qui impliquent des centaines, voire des milliers de règles de configuration. Dans cette thèse, nous proposons l'architecture d'un orchestrateur exploitant la programmabilité des réseaux pour automatiser la configuration et le déploiement de chaînes de fonctions de sécurité. Il est important que ces chaînes de sécurité soient correctes afin de éviter l'introduction de failles de sécurité dans le réseau. Aussi, notre orchestrateur repose sur des méthodes automatiques de vérification et de synthèse, encore appelées méthodes formelles, pour assurer la correction des chaînes. Notre travail appréhende également l'optimisation du déploiement des chaînes dans le réseau, afin de préserver ses ressources et sa qualité de service.

Mots-clés: Vérification Formelle, Sécurité des Réseaux, Orchestration de Services

Abstract

Smart environments, in particular smartphones, are the target of multiple security attacks. Moreover, the deployment of traditional security mechanisms is often inadequate due to their highly constrained resources. In that context, we propose to use chains of security functions which are composed of several security services, such as firewalls or antivirus, automatically configured and deployed in the network. Chains of security functions are known as being error prone and hard to validate. This difficulty is caused by the complexity of these constructs that involve hundreds and even thousands of configuration rules. In this PhD thesis, we propose the architecture of an orchestrator, exploiting the programmability brought by software defined networking, for the automated configuration and deployment of chains of security functions. It is important to automatically insure that these security chains are correct, before their deployment in order to avoid the introduction of security breaches in the network. To do so, our orchestrator relies on methods of automated verification and synthesis, also known as formal methods, to ensure the correctness of the chains. Our work also consider the optimization of the deployment of chains of security functions in the network, in order to maintain its resources and quality of service.

Keywords: Formal Verification, Network Security, Service Orchestration